编程狂人

Programming Madman

NO.51

推酷

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/admin/mag/periods/5473346ad91b141ea501834b/edit

欢迎下载推酷客户端体验更多阅读乐趣

目录

当 AMD 遇上 FIS

作者：2betop

前言

也许说 AMD 不知道这是啥，但说 requirejs 就都懂了。没错 AMD 就是一种模块定义的接口（API），用来定义模块间依赖以及自身暴露什么内容的一种规范。而 requirejs 就是一种实现了这些接口的 AMD Loader。

说到 requirejs 相信不少人都已经对它爱不释手了，它真是给我们的开发带来了不少便利性。只要我们每个模块都简单的遵守这个规则

```
// app.js
define(function (require, exports, module) {
    var a = require('a');
    var b = require('b');

    exports.action = function () {};
});
```

然后，简单一段

```
// 程序入口
require(['/app'], function(app) {
```

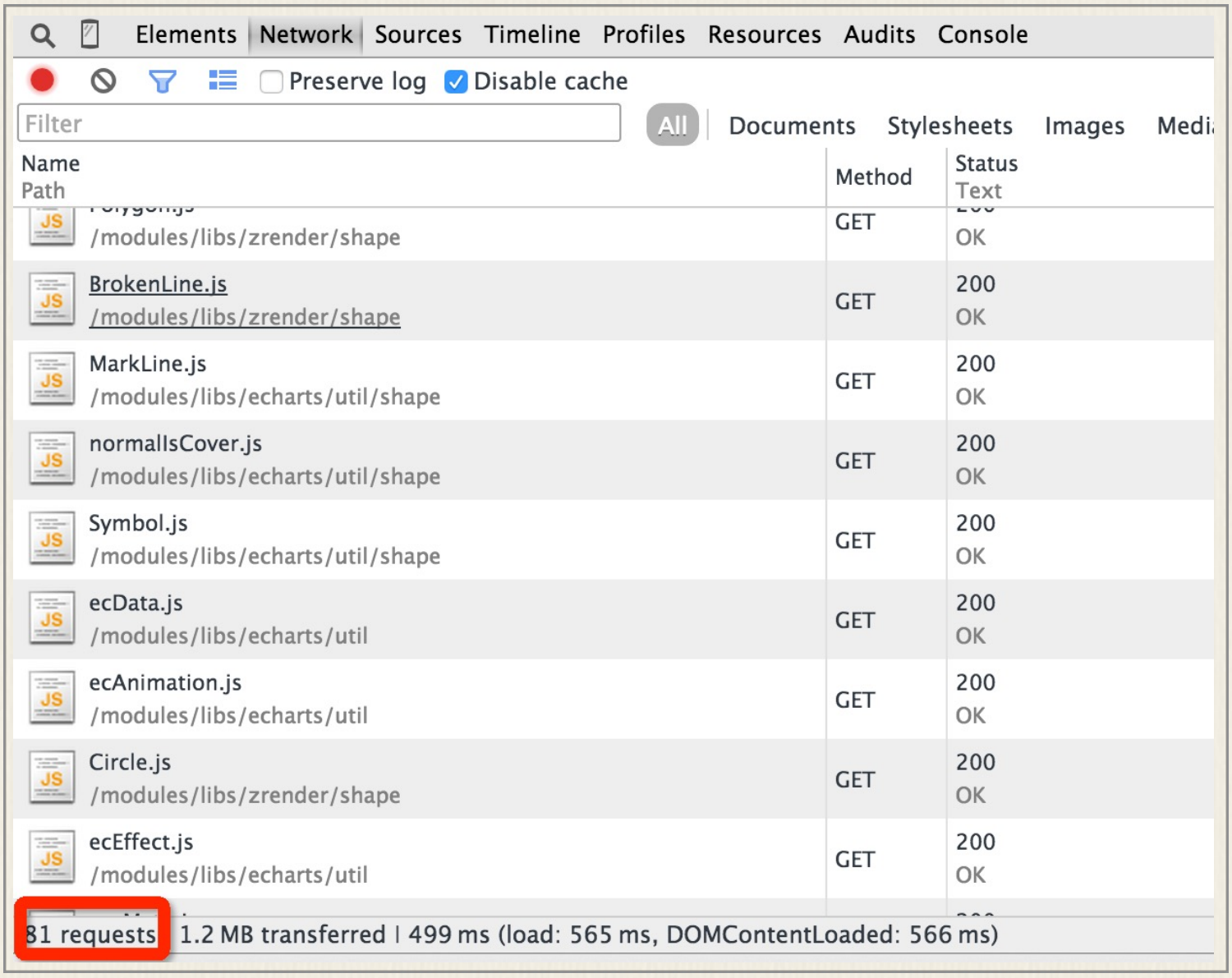
```
    app.action();
});
```

就把指定的所有依赖都自动加载进来了。

于是，我们慢慢的会把一个大功能模块，拆得非常小，让每一个模块都只干最少的事，而且我们很享受这样的拆分，因为这样带给我们非常棒的可维护性。

问题

当我们把代码拆得非常小之后，直接用 requirejs 去加载的时候，很容就会出现这种情况。



性能好不好，可想而知。 于是乎，我们需要把这些依赖打包起来。如何打包？当然是 r.js 他提供一种指定入口文件将所有的依赖打包成一个文件的工具。常用的做法是，配置一个列表给每个入口程序都打成一个文件, 然后手动把所有的入口文件地址换成打包后的。

这样基本上能满足需求，但是仍然还有些问题?

- 每个入口及其依赖打成了一个包，多个页面间公用的依赖被打包到了多处，页面切换公用依赖的缓存完全没有被利用起来。
- 每个入口地址我都得手动替换新地址，麻烦!
- 有些 amd 模块写法，需要 requirejs 在运行期需要将 function 转成字符分析依赖，性能会不会有问题?

优化方案

如果你使用 FIS, 这些问题就都迎刃而解，而且还能带来其他更多的好处。你可以先试用一下这个 fis amd demo。然后，让我让我来细说 fis 针对 amd 模块做了哪些优化以及在 fis中使用将带来哪些好处。

全新的编译插件

使用过 fis mod.js 方案的同学应该知道。原来对 js 模块依赖的解析只是简单粗暴的分析了两种用法。

即:

1. require('xxxx')
2. require.async('xxxx', cb);

将依赖生成 map.json, 然后，模块定义就是让用户去遵循 commonjs 规范，FIS 在编译期会自动封装成 amd module，其实就是包了一层 define

```
define(moduleId, funciton(require, exports, module) {
    // 源 JS 内容。
```

```
    var a = require('a');
    var b = require('b');

    exports.action = function() {};
});
```

在页面渲染的时候，程序会根据 map.json 中依赖的申明，提前把同步依赖加载进来。

其实这样已经满足各种开发需求了，而且非常高效实用。但是随着外界开源组件的兴起以及 bower 的推广，目前已有大量的第三方组件涌现，而且他们都有一个特点，就是采用的 amd 规范。问题就是，这些组件拿过来放在 fis 中，没法直接用，必须得手动修改才能使用。

于是，全新的 amd 依赖解析插件 诞生了。

它会分析所有 AMD 规范中定义的各种写法。

有了它，模块间的依赖实际上在编译期就已经知道了，并把的依赖关系生成到了 map.json, 这样只要借助工具，就可以提前把所需模块的全部依赖提前加载进来，而不需要让 requirejs 在前端用 js 去动态加载。

怎么让 requirejs 不重复加载？只要提前加载进来的模块，都带上 module id, 然后 require 入口引用的 module id 与之一致，requirejs 是不会重复加载的。这个自动补充 module id 的工作在这个插件中自动完成了，默认是自动把该文件在工程下面的路径去掉 .js 后缀的值作为 module id。

有了这些依赖信息，我们还可以利用 combo 或者 pack 打包 将依赖合成一个文件输出，这样就减少了多个请求带来的网络开销，以后可以愉快拆分模块代码了。

更好的模块化开发体验

一个大型的项目，一般情况都会包括三种类型的模块。

1. 第三方模块
2. 当前项目可公用的模块
3. 应用级模块，每个页面都不一样。

针对这三种性质的模块，我们都比较喜欢放在不同的目录。这样带来的坏处是，不管我用绝对定位还是相对定位，都是如此的别扭。

感谢 AMD 规范中制定了3个非常便于查找模块路径的配置, 我们把这几个配置也应用到了编译期。

通过fis.config.set('settings.postprocessor.amd') 来设置。

```
fis.config.set('settings.postprocessor.amd', {
    baseUrl: '.',

    // 查看：https://github.com/amdjs/amdjs-api/blob/master/CommonConfig.md#paths-
    // 不同的是，这是编译期处理，路径请填写编译路径。
    paths: {
        jquery: 'modules/libs/jquery/jquery.js',
        bootstrap: 'modules/libs/bootstrap/js/bootstrap.js',
        jqueryui: 'modules/libs/jquery-ui/ui/',
        app: './modules/app',
        css: './modules/css.js'
    },

    // 查看：https://github.com/amdjs/amdjs-api/blob/master/CommonConfig.md#packages-
```

```
// 不同的是，这是编译期处理，路径请填写编译路径。
packages: [

    {
        name: 'zrender',
        location: 'modules/libs/zrender',
        main: 'zrender'
    },

    {
        name: 'echarts',
        location: 'modules/libs/echarts',
        main: 'echarts'
        }
    ]
});
```

baseUrl

当设置了 baseUrl 后，所有绝对路径的模块查找都是基于此目录查找的，对于使用频率比较高的模块，可以把改目录设置成 baseUrl. 比如第三类模块。

paths

对于一些常用的库，可以通过这个来设置短引用或者说别名。比如：jquery, bootstrap。

另外需要说明的是，有些第三方库在发布的时候，都是指定的别名依赖。如： jquery-ui 一系列。这种模块有很多很多。

```
(function( factory ) {
    if ( typeof define === "function" && define.amd ) {

        // AMD. Register as an anonymous module.
        define([
            "jquery",
            "./core",
            "./widget",
            "./position"
        ], factory );
    } else {

        // Browser globals
        factory( jQuery );
    }
}(function( $ ) {
});
```

所以，为了不动第三方源码，我们也需要明确的设置这个别名。

```
paths: {
```

```
    jquery: 'modules/libs/jquery/jquery.js'
}
```

其次，我们可以给这类性质的“当前项目可公用的模块”，设置个 paths. 如:

```
paths: {
    libs: '/widget/libs/'
}
```

这样对于内部公共模块目录下模块的引用无论你的代码在什么位置就可以这样引用。

```
define(function(require, exports, module) {
    var dialog = require('libs/dialog');
    ...
});
```

packages

作用基本上和 paths 差不多，只是它更适合配置成一个完整的模块包。如 zrender、echarts 等等。

更智能的包装

在 FIS 开发环境中，你还可以编写满足 commonjs 规范的 js 模块，FIS 会自动包装成 AMD 模块以便于在浏览器中运行。如果直接就是 AMD 规范编写的，那就更不用说了。但是既不是 amd, 也不是 commonJS 规范的模块怎么办呢?

再次感谢 AMD 规范中的 shim 配置，同样，FIS 把这个配置应用到了编译期。

```
fis.config.set('settings.postprocessor.amd', {
    // 设置 bootstrap 依赖 jquery
    // 更多用法见：https://github.com/amdjs/amdjs-api/blob/master/CommonConfig.md#shim-
    // key 为编译期路径。
    shim: {
        'modules/libs/bootstrap/js/bootstrap.js': ['jquery'],

        'some/ohther/path.js': {
            deps: ['libs/a', 'libs/b'],
            exports: 'some.thing',
            init: function (a, b) {
                return some.thing + 'another';
            }
        }
    }
});
```

FIS 在包装模块组件的时候，会读取此配置，自动把改模块的依赖和暴露的对象添加上。当然 requirejs 本来也能做这个事，但是考虑到性能开销，这个工作更应该在编译期完成。

更高级的插件加载机制

AMD 除了可以处理 JS 模块依赖加载，还能处理其他依赖加载，怎么做？就是利用 amd plugin loader

换句话说 amd 还可以用来加载 css or 前端 tpl。在 fis amd demo 例子中有个示例，就是利用 css amd 插件来动态加载 css 文件。amd 依赖解析插件 不仅只处理模块查找，还处理插件资源查找。

```
require(['css!./styles/demo.css'], function () {
    document.getElementById('main').innerHTML = '<div id="demo">It works!</div>';
});
```

这样的好处是，对于当前工程下面的静态资源引用，可以用相对路径，也可以用绝对路径，且可以给资源加 md5 戳，甚至可以最终部署到 cdn 上，而不用改一句源码。

更智能的打包

FIS 的 pack 打包方案本来就比较灵活，通过正则或者 glob语法，可以把任意多的文件合并成一个。同时当使用 depscombine 插件的时候也支持 r.js 那种方式，将入口文件的所有依赖合并进来，只要在合并入口 JS 依赖前，配置一条规则把公用依赖部分的 js 合并成一个文件，就能把公共依赖抽离出来，这样公共的部分缓存就可以被利用起来。

```
fis.config.set('pack', {
    // 依赖也会自动打包进来, 且可以通过控制前后顺来来定制打包，后面的匹配结果如果已经在前面匹配过，将自动忽略。
    'pkg/zrender.js': ['modules/libs/zrender/zrender.js'],
```

```
    'pkg/echarts.js': ['modules/libs/echarts/echarts.js'],

    'pkg/bootstrap_jquery.js': ['modules/libs/bootstrap/js/bootstrap.js'],

    'pkg/jquery_ui_tabs.js': ['modules/libs/jquery-ui/ui/tabs.js']
});
```

Name Path	Method	Status Text	Type
index.html	GET	200 OK	text/html
require.js /lib	GET	304 Not Modified	application/javascript
jqueyr-ui.css /pkg/css	GET	200 OK	text/css
page_map_1.js /pkg	GET	200 OK	application/javascript
bootstrap_jquery.js /pkg	GET	304 Not Modified	application/javascript
jquery_ui_tabs.js /pkg	GET	200 OK	application/javascript
ui-bg_flat_75_ffffff_40x100.png /modules/libs/jquery-ui/themes/base/images	GET	200 OK	image/png

当配置好规则后，简单的一个 fis release -p 命令就把所有被打包文件的请求变成合并后的了，源码什么都不用改，如果想愉快的调试代码，release 时不带 -p 参数，又自动变成了非打包方案了。

被遗忘的技术细节

现在 require 入口调用，会自动把其同步依赖加载进来。但是，等等，貌似怪怪的，因为 require 入口的调用其语义就是异步调用，怎么变成同步的语义了?

按语义来应该针对 require('deps') 引用做同步处理，但是这种用法并不在 amd 规范中定义，amd 规范定义的同步调用用法，只出现在模块定义内

部。所以没办法，把模块定义外的 require 用法当成同步来用吧（模块定义内部的 require 异步语义保持不变）。当然一定要当作异步来用也是可以的，只要在 require 调用的前面加段注释 fis async。这样编译期就会把找到依赖标记成异步依赖。

由于 FIS 对于静态文件是支持打包合并、加 md5 戳和部署到 cdn 的，也就是对于 js 的引用，我们是要忽略他的 release 后的路径的。如果纯同步依赖，似乎没问题，但是异步依赖怎么办呢？我在 require 里面的 module id 当然还是得用源码路径ID方便调试定位。

那么怎么转换路径呢？

原来 mod.js 方案是读取 map.json 生成一个异步所需的 resoucemap 表，通过 require.resourceMap({xxx}) 设置给 mod.js，这样在异步加载模块的时候，可以对应找到实际的存放地址。

amd 方案里面也是采用同样的方式，只是利用的是 amd 规范中的 paths 设置，根据 map.json 自动生成程序中所需要的异步依赖的路径转换规则，这样的话，fis 不是一定只能用 mod.js 才能做模块化开发，只要满足 amd 规范的所有 loader 都能支持，比如 ecom 出的 esl.js;

```
require.config({"paths":{
    "modules/libs/zrender/lib/excanvas": "/pkg/zrender",
    "modules/libs/zrender/tool/util": "/pkg/zrender",
    "modules/libs/zrender/config": "/pkg/zrender",
    "modules/libs/zrender/tool/log": "/pkg/zrender",
    "modules/libs/zrender/tool/guid": "/pkg/zrender",
    "modules/libs/zrender/tool/env": "/pkg/zrender",
    "modules/libs/zrender/tool/event": "/pkg/zrender",
    "modules/libs/zrender/Handler": "/pkg/zrender",
    "modules/libs/zrender/tool/matrix": "/pkg/zrender",
```

```
    "modules/libs/zrender/shape/mixin/Transformable": "/pkg/zrender",
    "modules/libs/zrender/tool/color": "/pkg/zrender",
    "modules/libs/zrender/shape/Base": "/pkg/zrender",
    "modules/libs/zrender/shape/Path": "/pkg/zrender",
    "modules/libs/zrender/tool/area": "/pkg/zrender",
    "modules/libs/zrender/shape/Text": "/pkg/zrender",
    "modules/libs/zrender/shape/Rectangle": "/pkg/zrender",
    "modules/libs/zrender/loadingEffect/Base": "/pkg/zrender",
    "modules/libs/zrender/shape/Image": "/pkg/zrender",
    "modules/libs/zrender/Painter": "/pkg/zrender",
    "modules/libs/zrender/shape/Group": "/pkg/zrender",
    "modules/libs/zrender/Storage": "/pkg/zrender",
    "modules/libs/zrender/animation/easing": "/pkg/zrender",
    "modules/libs/zrender/animation/Clip": "/pkg/zrender",
    "modules/libs/zrender/animation/Animation": "/pkg/zrender",
    "modules/libs/zrender/zrender": "/pkg/zrender",
    "modules/libs/zrender/shape/Circle": "/pkg/echarts",
    "modules/libs/zrender/tool/math": "/pkg/echarts",
    "modules/libs/zrender/shape/Ring": "/pkg/echarts",
    ...
});
```

另外，amd 依赖解析插件 除了解析依赖，实际还会做一个小优化，就是会把 factory 中的各种依赖，提前放置在 define 的第二个参数里面。这样的

好处是， amd loader 再也不需要用正则查找 factory 函数体的 require 了，直接读第二个参数就能把所有依赖拿到。

总结

既然 fis 在编译 amd 模块的时候，优化了这么多，依赖处理啊， ID 生成啊之类的。那么我们还需要一个如此庞大的 require.js 吗？ 当然不需要，FIS 组结合编译的处理，提供一个最小 amd loader 叫 mod-amd.js 仅仅 200 多行, 但是他暂时不支持 amd plugin loader，因为没有足够的理由要去支持它，像模板加载，样式加载，fis 中有更优的处理方案。

好吧，回头正视原来提出的那些问题。

- 每个入口及其依赖打成了一个包，多个页面间公用的依赖被打包到了多处，页面切换公用依赖的缓存完全没有被利用起来。

采用 fis pack 打包配置，很好的解决这个问题。

- 每个入口地址我都得手动替换新地址，麻烦！

在 fis 里面编译的时候加上 -p 就足够。

- 有些 amd 模块写法，需要 requirejs 在运行期需要将 function 转成字符分析依赖，性能会不会有问题?

编译期，自动将依赖前置。

原文链接：http://fex.baidu.com/blog/2014/11/when-amd-meet-fis/

你可能不知道的10个CSS3中的隐藏特性

译者：shoothao

概述：W3C正不断致力于为设计师、开发人员和用户开发新的CSS特性，最新的CSS3为web设计增添了许多令人惊叹的特性，下面我们就来看看你可能不知道的CSS3中的10个隐藏特性.

CSS3 为web设计增添了许多令人惊叹的特性，这其中你经常会用到box-shadow（图层阴影），border-radius（边框圆 角），transform（变形）这一类受欢迎的常用特性。但是还有一些强大的功能，你可能没有接触到，它们就象是埋藏在地下的宝藏，静静等待着你的发 掘。

W3C正不断致力于为设计师、开发人员和用户开发新的CSS特性，下面我们就来看看你可能不知道的CSS3中的10个隐藏特性：

1.Tab尺寸控制

大多数代码编辑器配有Tab尺寸控制，开发者可以对指定代码的缩进所使用的Tab键的宽度进行控制。而现在，这个功能已经对嵌入在网页的定制代码开放了。

```
01.  pre {
02.      tab-size: 2;
03.  }
```

注意，每个浏览器可能都有对Tab占用宽度的不同说明。所以，我们在不同的浏览器上会看到一些差异。我目前所知道的支持这个功能的浏览器包括：Chrome，Opera，Firefox和Safari。

2.文本渲染

文本渲染的特性可以让浏览器知道如何在网页中渲染文本。文本优化所涉及的性能、易读性、精度将会决定到文本的质量。看看下面截图两个版本的字距调整，来辨别一下“正常”版和“易读性优化”版的差别吧。

3.字体伸缩

除 了常规的正常，粗体和斜体，其它的字体设置也会提供不一样的感官设计。Helvetica Neue或者Myriad Pro字体就是其中的例子，对于字体的伸缩程度，它们有“Condensed”，“Ultra-condensed”，和“Semi-condensed”几种设置，这些都可以通过一个名为font-stretch（字体伸缩）的特性来实现。

我们可以使用字体伸缩结合文字特性（比如使用字体样式），举一个例子:

```
01.  h1 {
02.      font-style: bold;
03.      font-stretch: ;
04.  }
```

font-stretch（字体伸缩）特性目前只能在Firefox和Internet Explorer 9（及以上版本）使用。

4.文字溢出

文字溢出特性指定了容器中溢出或者被截断内容的呈现方式。默认的处理方式是截断，被截断的内容会被隐藏起来。你可以设置用省略号来代表被截断的文本或者进行省略。如下图所示：

```
.content-box {
    text-overflow
}
```

正如你所想的，末尾三个点的省略号代表了被省略的内容。

5.书写模式

并不是每一种语言都是从左到右的书写的，有一些语言是从上到下的比如日语，还有一些语言是从右到左的比如阿拉伯语和希伯来语。

为了适应这些语言，CSS引入了名为书写模式的新特性来帮助开发人员改变内容的书写方向。例如，这个代码片断就是让本文书写的内容保持从左到右的顺序（无论什么语言）

```
p {
    writing-mode: rl-tb;
}
```

改变文本内容的顺序为从上到下，这可以通过设置为vertical-l,r来实现：

```
p {
    writing-mode: vertical-lr;
}
```

6.指针事件

pointer-events（指针事件）属性可以让开发人员控制鼠标指针在拖下，悬停和点击等事件下的行为。使用下图的命令后，指针点击链接将不会产生任何效果，链接会被完全禁用，而href标签中所指定的地址位置也会成为摆设。

```
a {
    pointer-events: none;
}
```

关于指针事件的一些关键问题将会在下一个版本CSS4中得到解决。

7.图片定向

在Photoshop等图片编辑器里，你可以通过旋转或翻转等改变图片的方向。现在，CSS3中的image-orientation（图片定向）功能可以让你可以对网页上的图片完成同样的处理。这里是一个如何让图片进行水平翻转的例子：

```
img {
    image-orientation: flip;
}
```

你可以通过恢复镜像（from-image）来保留原来图片的方向：

```
img {
    image-orientation: from-image;
}
```

8.图片渲染

类 似于文本渲染特性，图像渲染定义了网页上的图片质量，特别是可改变图片的大小。这个特性是由一堆数值组成的，不同的浏览器对于这些数值的设定都不一样。比 如，crisp-edges这个用于保存图片对比度并可预防图片边缘模糊的特性在Chrome极速浏览器里会使用webkit-optimize-contrast，而在IE浏览器里会使用nearest-neighbor。

```
img {
    image-rendering: crisp-edges;
    image-rendering: -webkit-optimize-contrast;/* Webkit */
    -ms-interpolation-mode: nearest-neighbor;  /* IE */
}
```

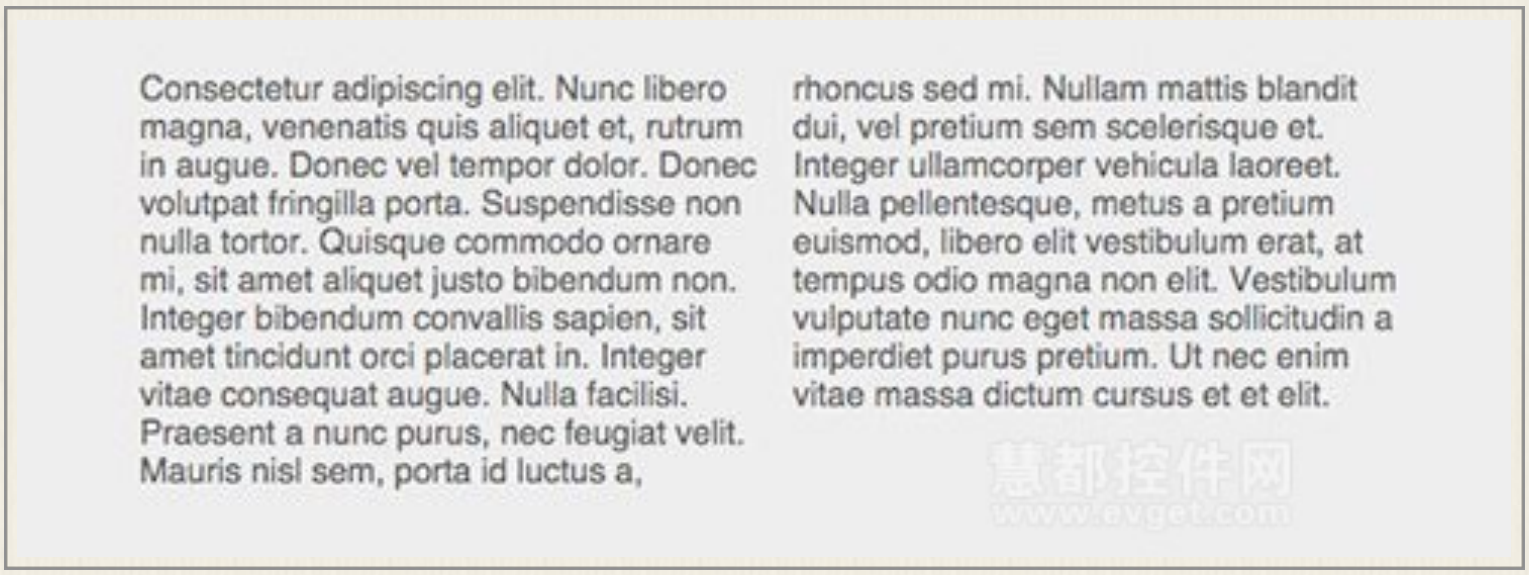
这是一项实验性技术，实施得到的结果有随着技术的不同而产生微小的变化。

9.列属性

列属性可以使开发者轻松的把web内容排列成列，我们把内容分成两列，如下所示：

```
.content {
    columns: 2;
}
```

当浏览器支持这个特性的时候，比如在Chrome和Safari中，我们就会看到以下内容的排列：



CSS这个特性上再加上一些创造力，你就可以轻松的创建一个与时尚杂志相似的拥有灵活流体布局和诱人内容布局的网站了。

10.flex流动布局

flex的特性旨在构建更加无缝化的响应式网格并同时解决关于主流网络布局使用浮动属性所产生的一系列问题。除此之外，使用flex特性，网络布局将完全延伸至整个容器，这在以前是一件相当烦琐的事情。

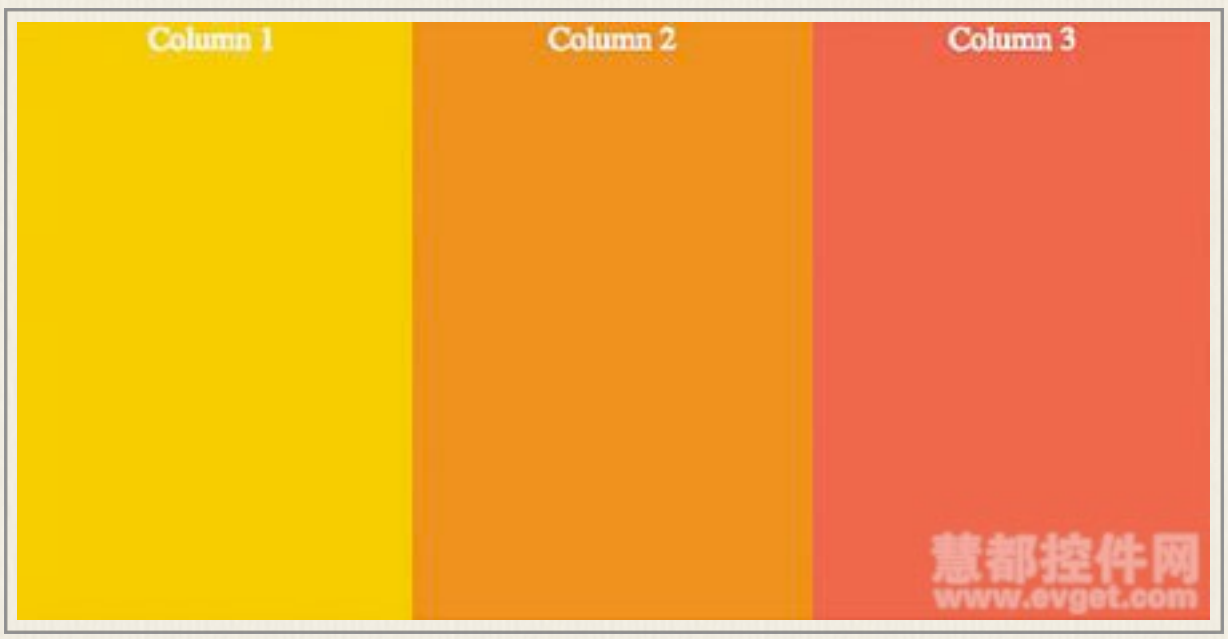
现在，假如你想要构建一个三列的web布局，你可以这样安排：

```
<div id="container">
    <div class="col">Column 1</div>
    <div class="col">Column 2</div>
    <div class="col">Column 3</div>
</div>
```

然后，使用flex构建列属性，像这样操作：

```
#container {
    width: 600px;
    height: 300px;
    display: flex;
}
#container .col {
    flex: auto;
}
```

附加字体和背景颜色的装饰，我们将得到以下结果：



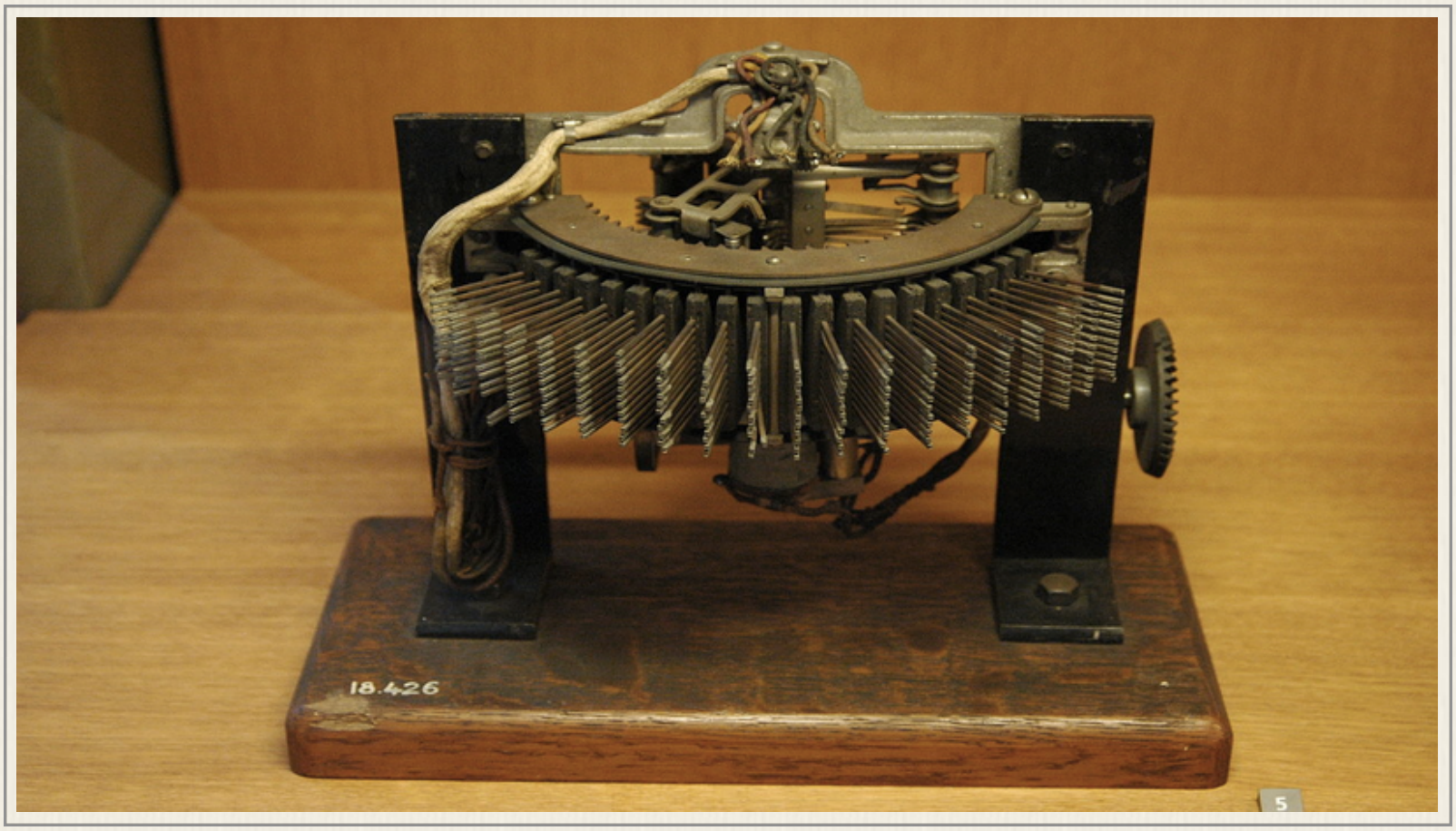
原译文链接：http://shoothao.iteye.com/blog/2158785

原文链接：http://www.hongkiat.com/blog/css3-properties-need-to-know/

缓存是新的内存

译者：douxingxiang,stefanzhlg

这是一次在 defrag 2014的演讲。



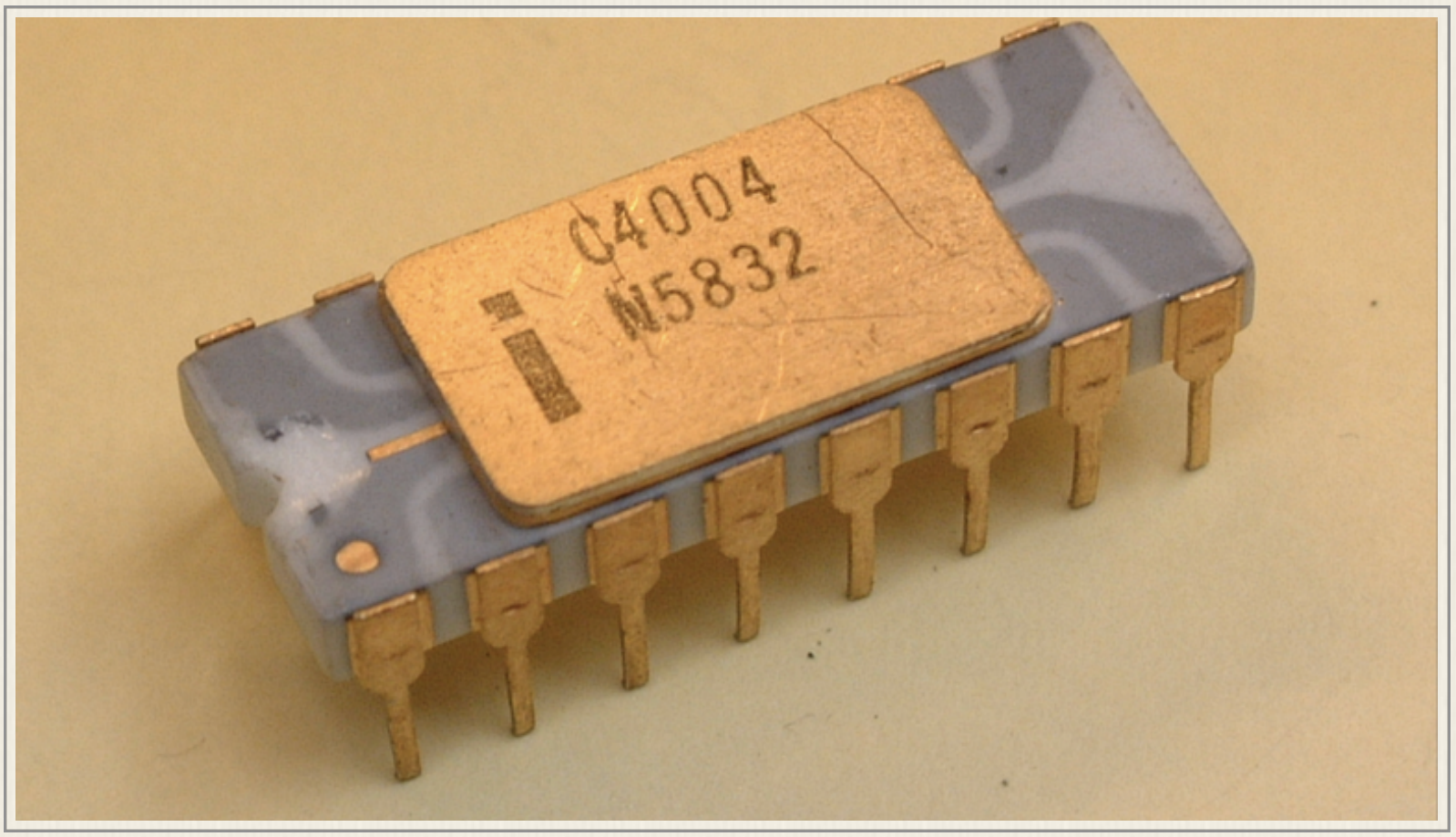
这是经过长时间地多次技术变革后的（多个）技术优势之一。你看到了实际上突破。如果你只是看到了其中的一部分，很难正确推断。你要么短期有进展，要么落后 很远。令人惊讶的不是事物变化的速度，而是一点一滴长期工程实践的突破。这是史端乔交换机，一个自动连接电话线路设备，在1891年发明的。

1951年，正是转向数字交换技术之时，一个典型的集中式交换中心基本上还是维多利亚时期的技术的放大版。每个转接过来的电话都有自己单独的strowger交换机。

当时来看，这已经是最牛逼的技术。当然我们看来，这只不过是当时世界上最大的蒸汽朋克(Steampunk，背景设在19世纪的科幻小说)风格的装置艺术(art installations)。

对此感到优越感可能是不对的。虽然集成电路(integrated circuit)已经面世65年了，仍然有数亿计的这种设备嗡嗡咔咔地运行着。直到现在，我们才真正地处在完全电子计算(solid-state computing, solid-state与机械相对，指基于半导体的)的转折点。

最令人兴奋的技术转变，一个是新的模型成为可行，另一个是旧的限制不再存在。在我们的工业界，这两种类型的转变都在上演。

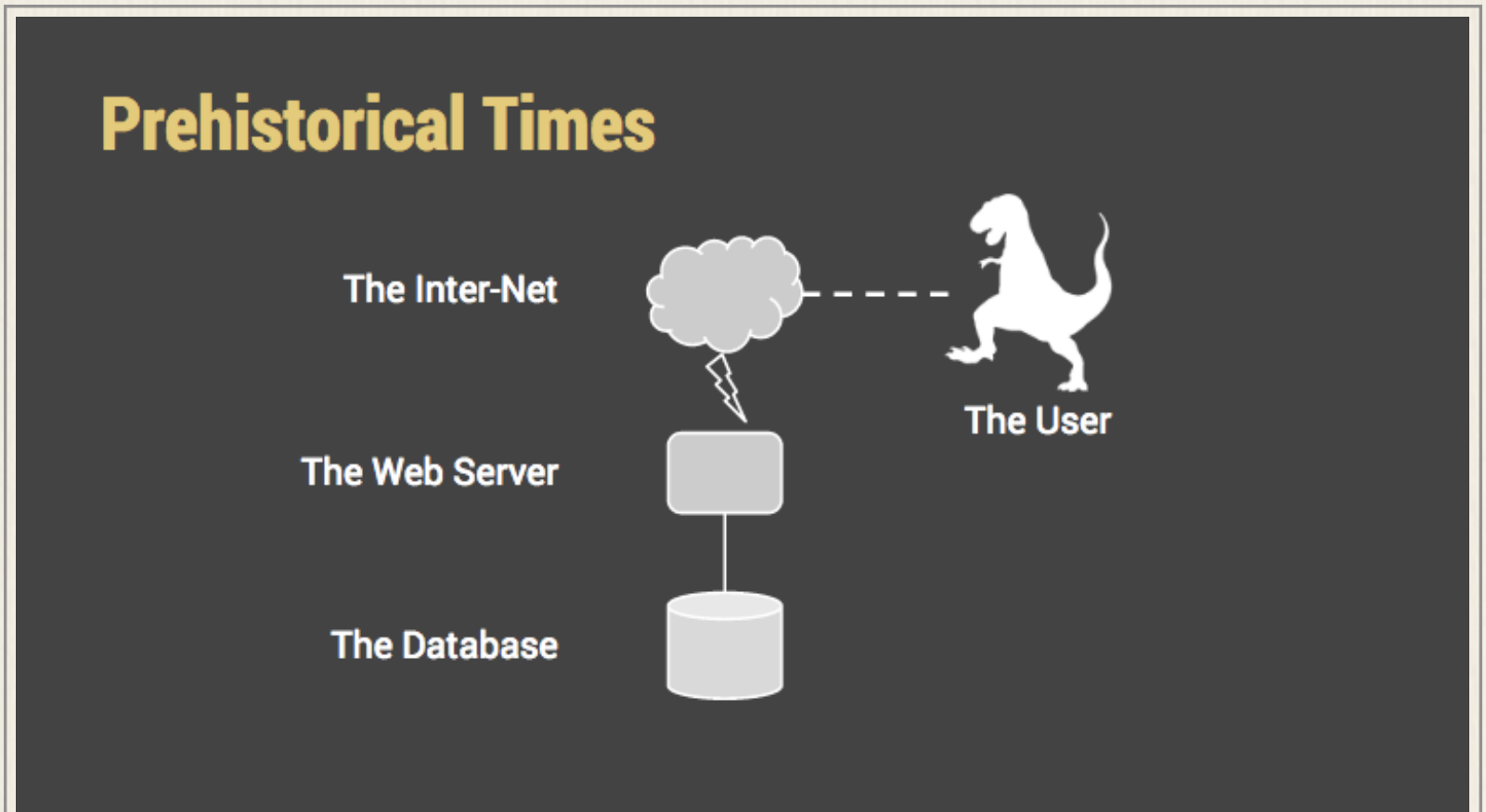


分布式计算(distributed computing)现在是贯穿整个软件栈的主导性的编程模型。所谓的中央处理单元(central processing unit)不再是中心化的，甚至都不是一个单元了。它仅仅算是数据之山(a mountain of data)上爬行的一群虫子(Bugs)中的一个。数据库是最后的堡垒。

CPU Register	1 ns
Main Memory	100 ns
Flash Drive	100,000 ns
Hard Drive	10,000,000 ns

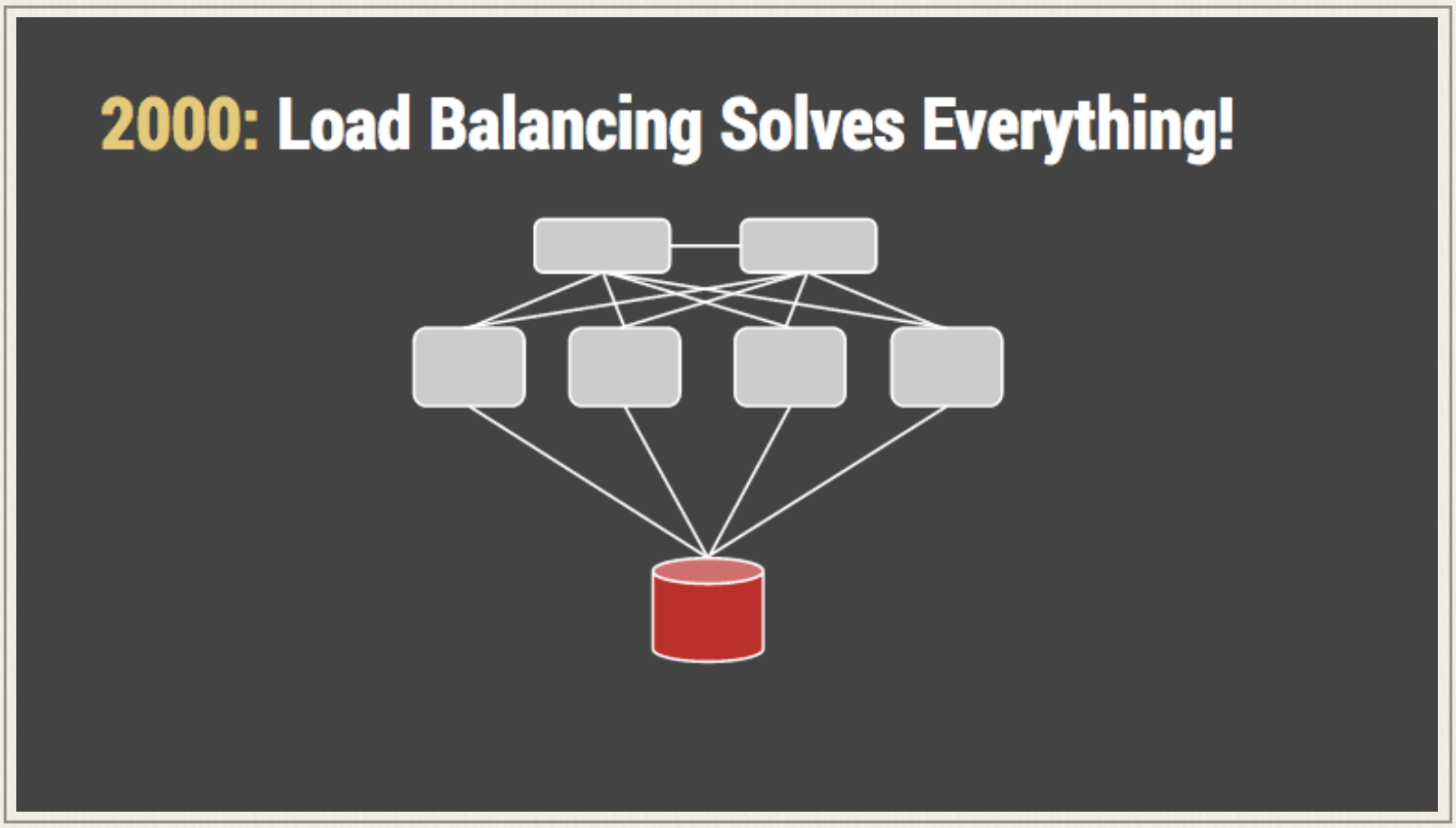
同 时，内存与硬盘存储间的延迟正在变得无关紧要。30年来，数据库性能的主要关心点，在于访问内存与硬盘存储上的随机数据的巨大差别。既然现在我们可以把数 据全部放在内存中，这些烦恼统统不用考虑了。当然不是这么简单，你不能用一个B树，mmap一下，然后就能搞定。在完全基于内存的设计方案推出之前还有很 多相关的东西要解决。

这两种新趋势产生了完全崭新的方式来思考、设计、构建应用。现在我们来谈下我们怎么达到，我们怎么做的，未来给我们的启示。



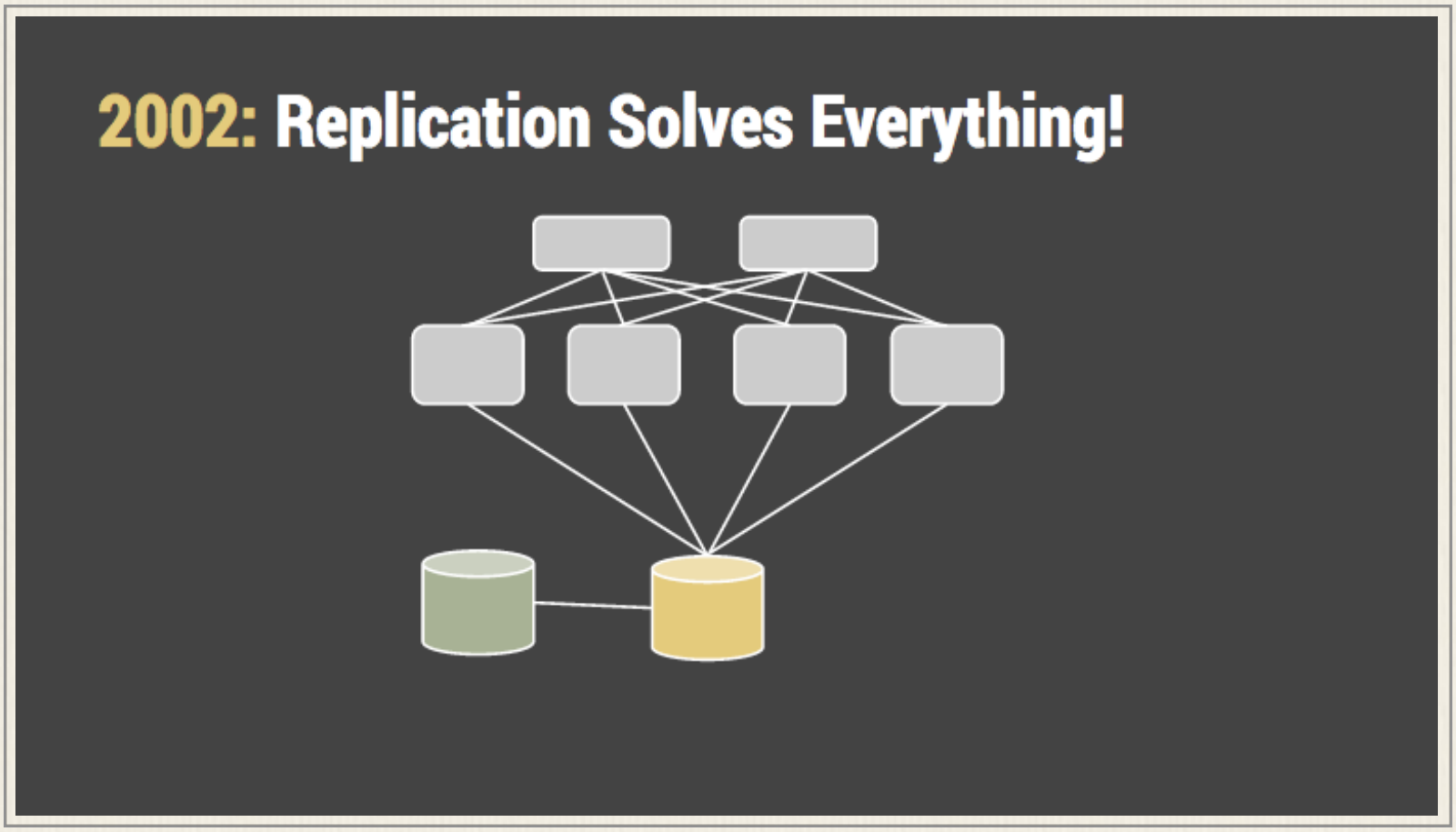
(史前时代，从下文看应该是2000年前，用户被描述成恐龙，作者的小幽默)

那时候，架构图里的每个组件都有一个确定的描述与之相关。每个组件都是一个单独的功能：数据库、web服务器，都成为一屋之剧中的不同角色(一屋指的是机房或data center)。 顺带提一下，这就是“the cloud”这个词的来源。一个轻软/ 毛茸茸的云是WAN的标准符号，而WAN的细节我们完全不用操心。



(2000年，负载均衡解决一切)

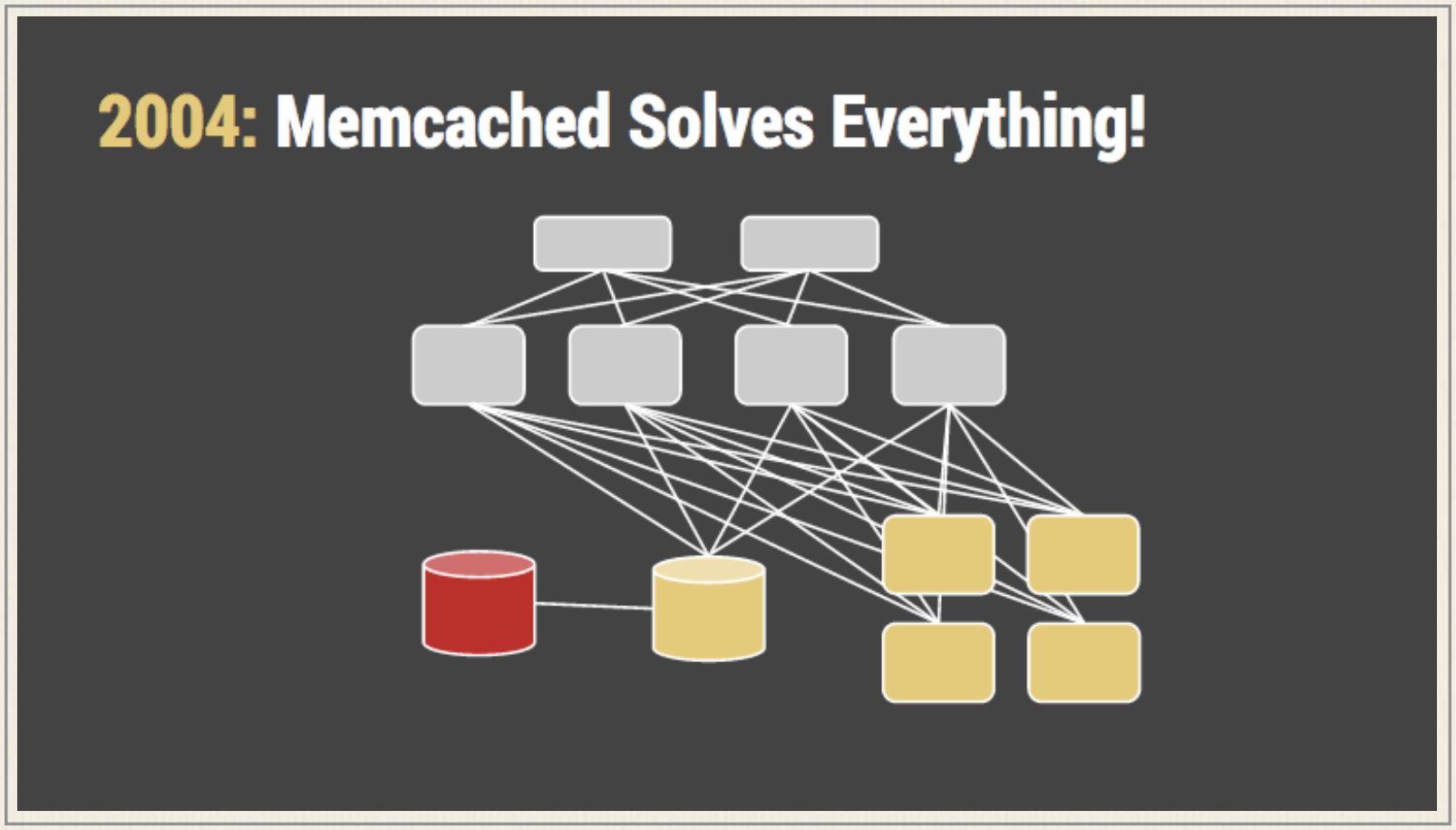
容易实现的分布式计算得到了主流的亲睐。多个完全相同的应用程序服务器藏在一个负载均衡器(load balancer)之 后，这个均衡器把负载差不多平均地分配到应用程序服务器上去。只负载均衡那些架构中状态无关的部分回避了很多哲学上的问题(理论上的情况？)。当系统扩展 时，这些组件从侧翼包抄，最后包围了“the” database。我们告诉自己，给数据库换上更快的磁盘、更快的CPU很正常，毕竟还是只需要一台机器。硬件提供商很高兴地赚着我们的钱。



(2002：备份解决一切)

最后，数据库备份成为合情合理的，加了一个热备份数据库(hot spare database)后，我们的良心得到些许宽慰。然后我们告诉自己，不会再有任何故障了。当然，这种正确性只存在了几分钟。

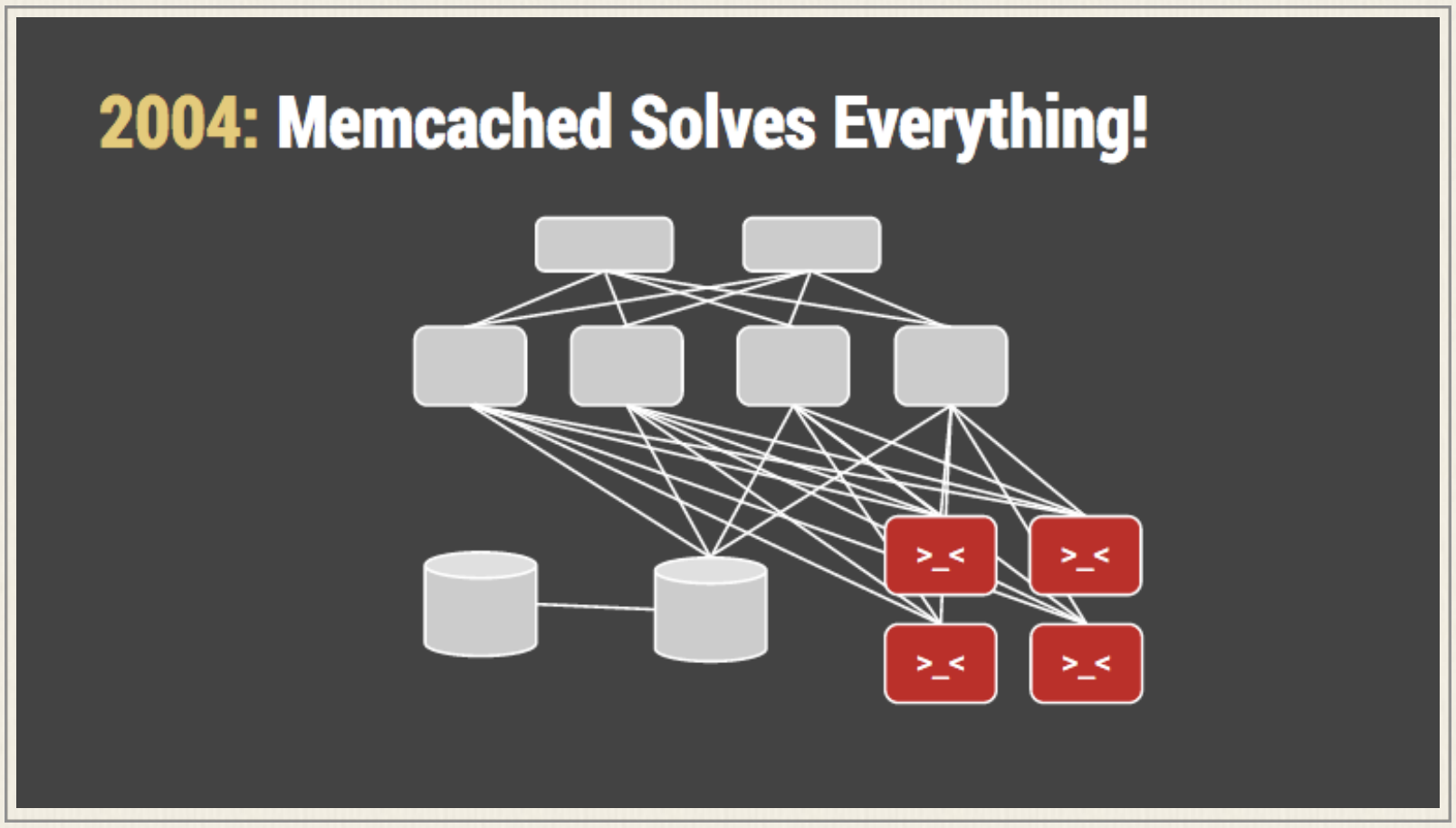
当 然，热备份经常是空闲的(sitting idle)。一旦商业分析员意识到，他们可以在不触及生产数据的情况下，也能对生产数据进行大规模查询，那么所谓的热备份也几乎跟生产数据一样开始忙碌并 且至关重要了。我们又告诉自己，在紧急情况下把热备份暂时拿出来也还好。但这就如同说，我们完全没必要带备胎，因为我们可以从其他车上偷一个过来！



(2004：memcached/缓存解决一切)

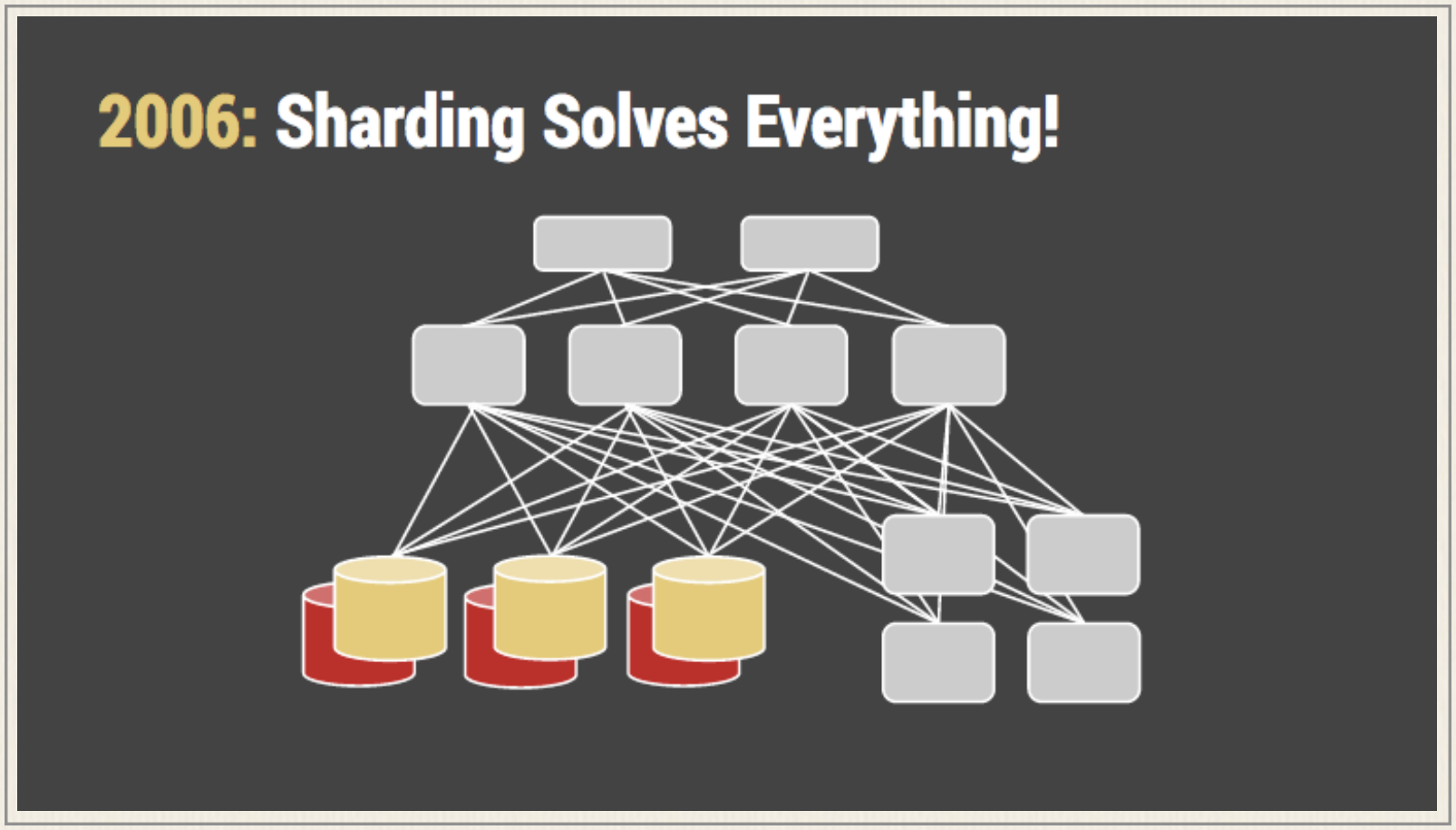
然 后，Brad Fitzpatrick发布了memcached，一个可以在内存中缓存数据的守护程序(因此叫memcached, memory cached)。这是个简化版的分布式哈希表，而且确实非常实用，因而之后就在学术界流行起来。它拥有很多特性：备份(a form of replication?)，水平分区，负载均衡，简单数学运算等。 我们再次告诉自己，既然大部分的负载都是读，我们为什么还要催促数据库从磁盘一遍一 遍做相同的查询？你只需要一组内存很大的小规模(small-calibre,小口径)服务器，当然硬件提供商也高兴地赚我们(买内存)的钱。

也许需要你写些缓存失效(cache invalidation)代码，这听起来不难，对吧。



(2004：memcached解决一切，添加了缓存失效)

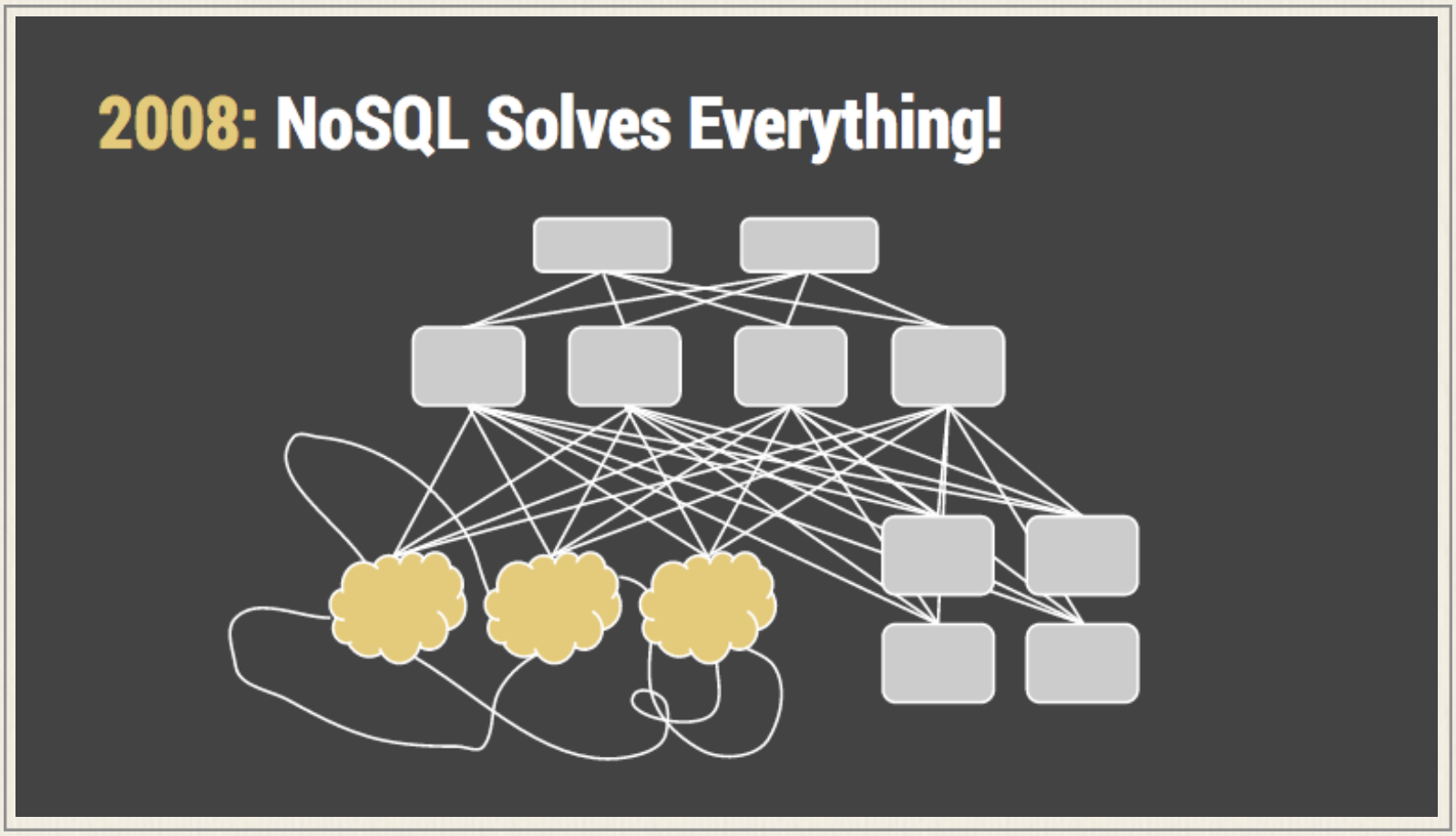
确 如其声称，memcached的方案使我们受益很长时间。它把硬盘的随机IO操作替换为内存的随机IO操作。尽管如此，那台数据库机器还是越来越大，越来 越忙。我们意识到，缓存的内存开销至少会跟工作集一样大(不然就是无效了)，再加上让人不能忍的缓存持久化。我们告诉自己，这就是网络级规模(web scale？)的开销。



(2006：水平分区/切分解决一切)

更 令人担心的是应用越来越复杂，越来越聊天化(chattier，可能聊天程序对数据库写的次数很多)。几乎每次都会进行多次数据库写操作。现在，写，而不 是读，成为了瓶颈。这时我们才最终认真对待数据库切分。Facebook最初是根据university字段来切分其用户数据，然后做成了"哈佛数据库 (The Harvard Database)"，并且维持了很长一段时间。Flickr是另一个好例子。他们使用PHP手动建立了一个切分系统，这个系统使用用户ID的哈希值来切 分数据库，跟memcached根据key来切分很像。在技术交流会上，他们透露，不得不对数据表去规范化(denormalize)，以及对一些对象(比如评论、消息、喜欢)进行两次写(doule-write)。

要解决无限伸缩(infinite scaling)总要付出点代价，对吧。

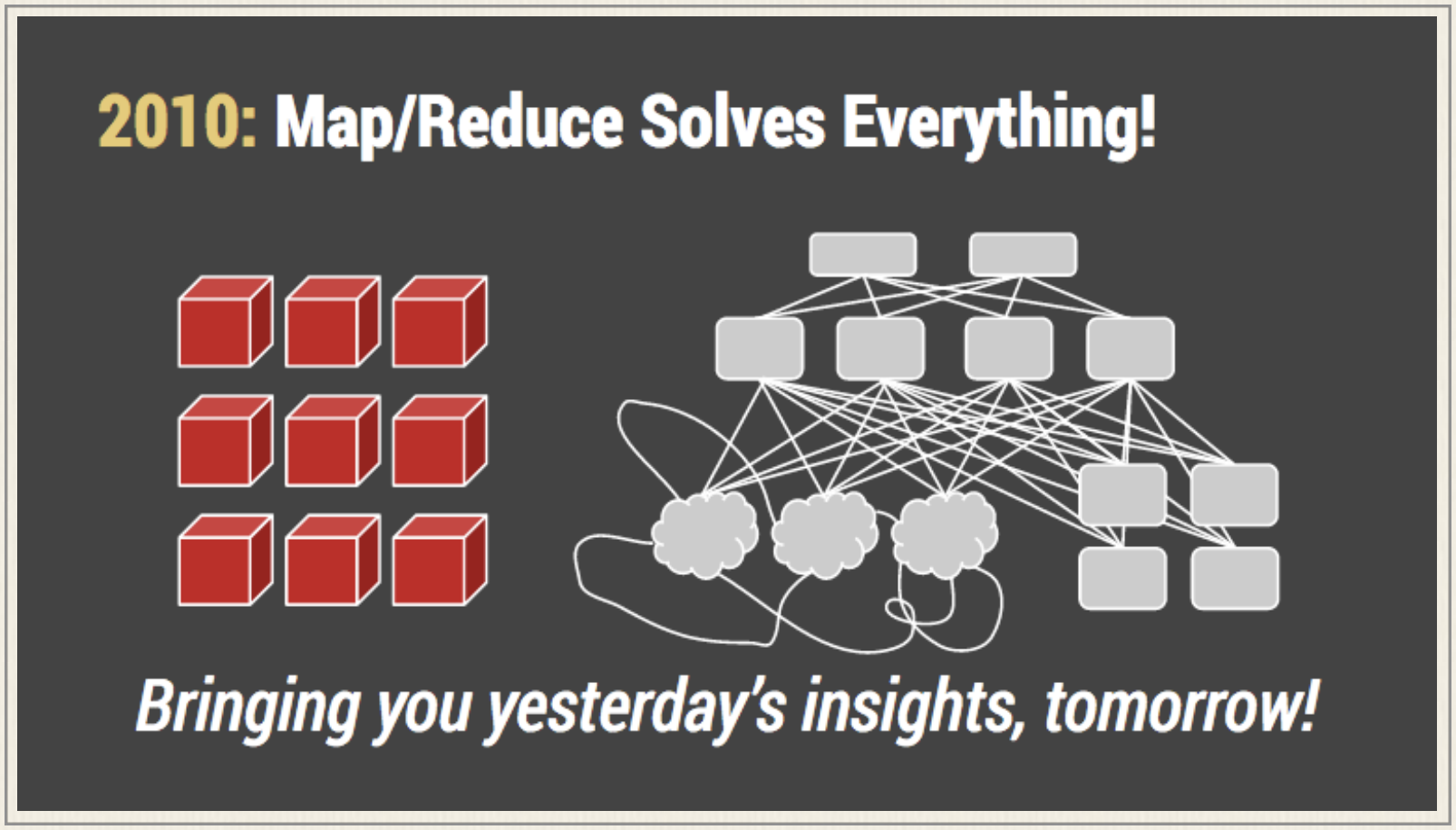


(2008：NoSQL解决一切)

手动切分关系型数据库的问题是，你的关系型数据库已经没了。切分API实际上成为你的查询语言了。你对操作的头疼还没好，而修改一组模式(schema)更加痛苦。

这 就需要大家深呼一口气，列出大家选用的SQL实现的所有不足和瑕疵，然后因此责怪SQL。一波潮人似的NoSQL，难民似的XML数据库出现了，并且都作 出了根本办不到的承诺。它们提供了自动切分，灵活的模式，一些冗余，...，一开始也就这么多。但是总比自己写要好多了。

你知道，“不用自己写”成为主要卖点的东西总是令人绝望。

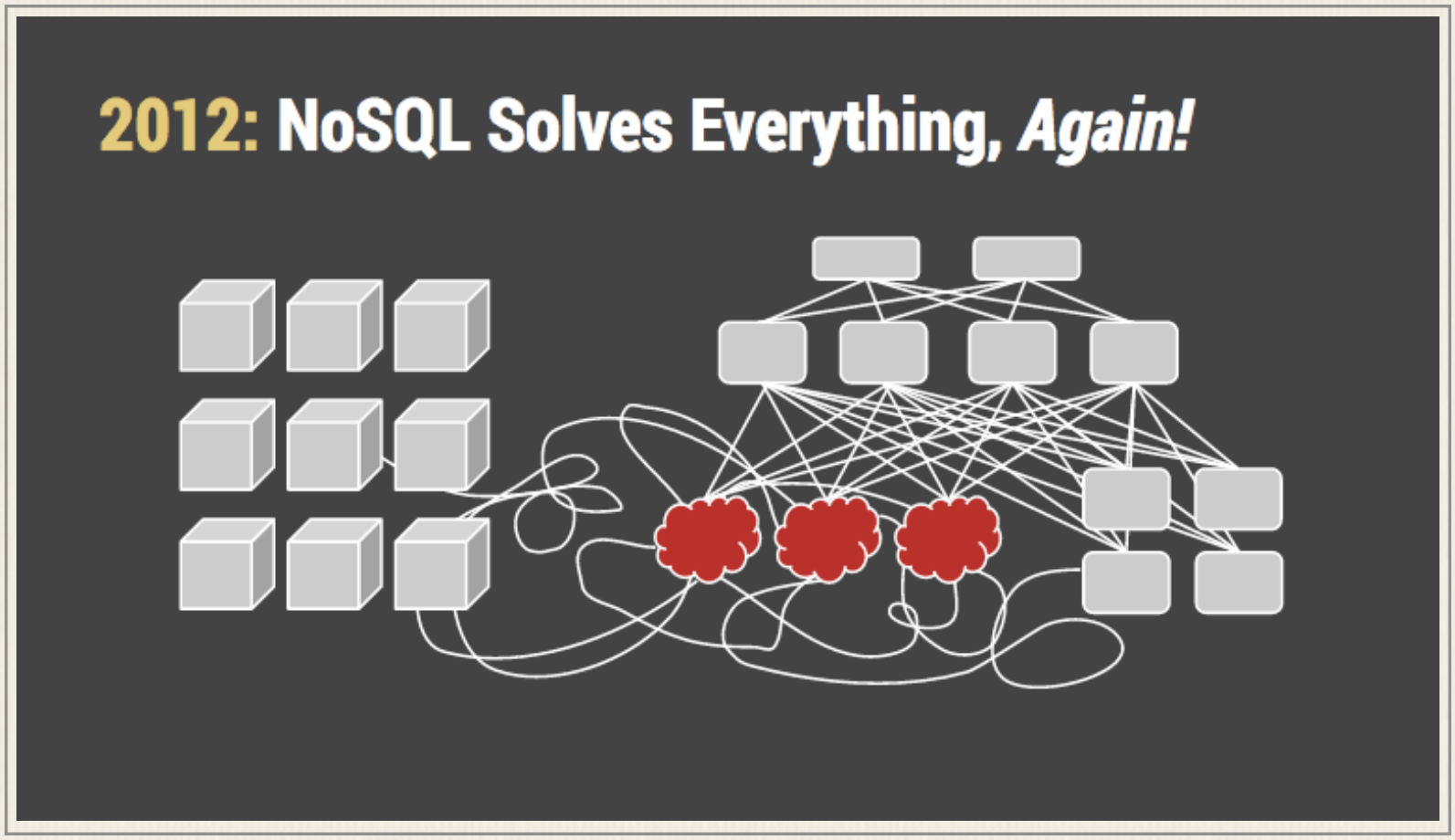


(2010：Map/Reduce解决一切)

转移到NoSQL并不比使用手动切分差，因为我们已经放弃了使用常用的客户端工具控制和分析数据的希望。但这没好多少。之前由商业人员(business folks)编写的SQL查询变成了开发人员维护的报表代码。

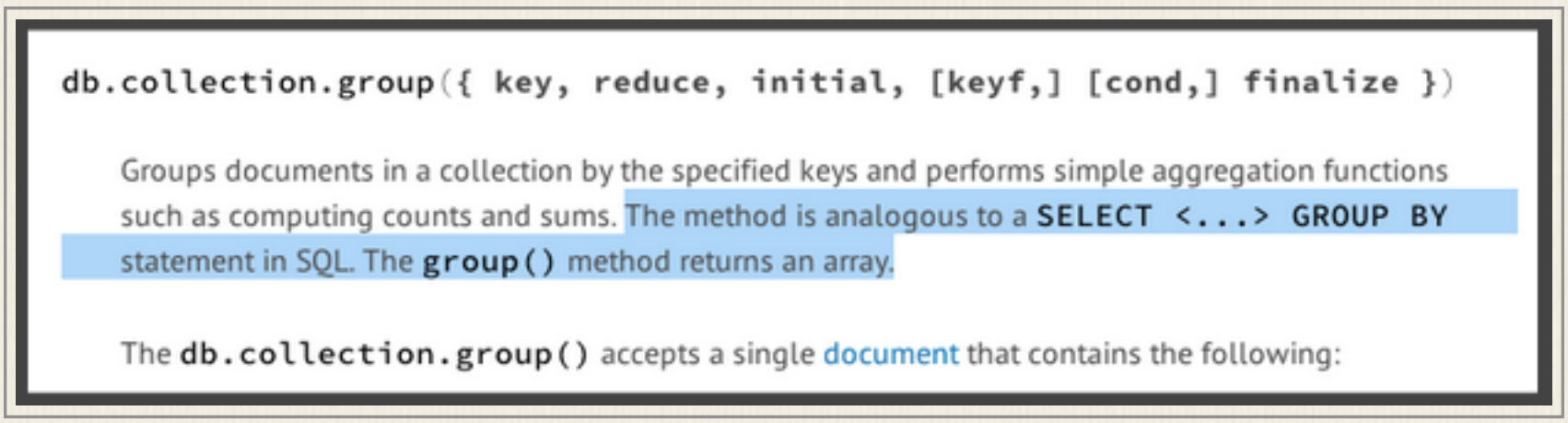
还记得用于备份和分析的热备份数据库吧？现在它变身为Hadoop filestores以及上层的Hive查询而卷土重来了。既然奏效，商业人员再也不来烦我们了。但一个大问题是，这些系统的操作复杂性。就像航天飞机一样，它们是作为可靠且几乎不用维护的产品出售的，但是最后还是需要大量的手动操作。另一个大问题是，数据的存入和取出：花费一整天的时间已经相当不错了。第三个大问题是IO同时成为网络和磁盘的瓶颈。我们告诉自己，这就是从大数据(big data)毕业的代价。

不管怎样，Google就是这样做的，对吧。



(2012：NoSQL再次解决一切)

随着一些NoSQL数据库的逐渐成熟，它们的API发生了诡异的变化：它们开始长得像SQL一样。这是因为SQL是关系型集合理论(relational set theory)的相当直接的实现，而数学不是那么好愚弄的。



我重述下Paul Graham对Lisp 那难以忍受、并自鸣得意的评论：一旦你添加了group by, filter, join，你也不能声称发明了新的查询语言，因为这仅仅算是SQL的一个新方言。而且语法很差，还没有优化器。

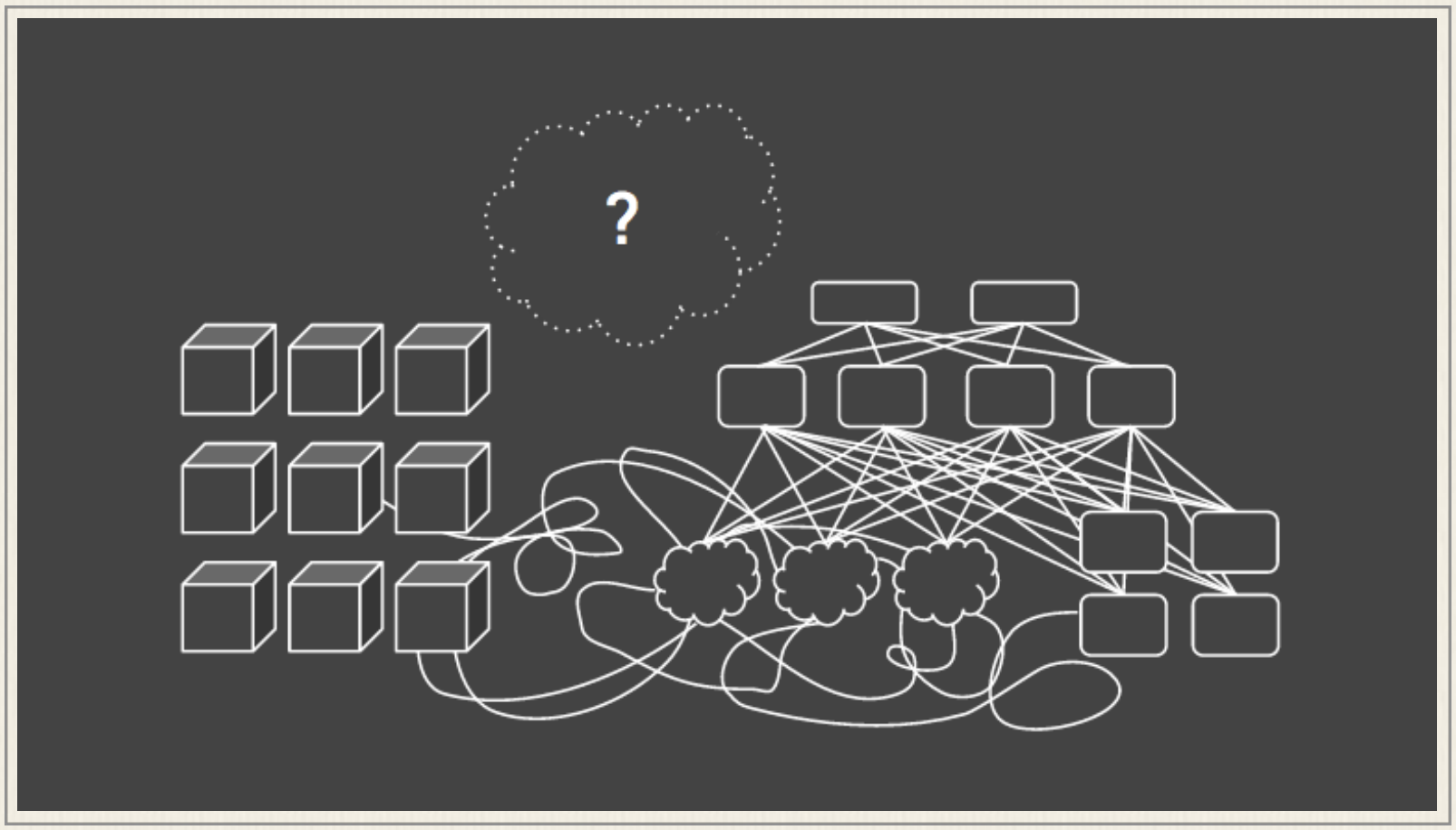
由于我们绕过了SQL，大部分系统都缺少了一些很重要的东西，比如存储引擎、查询优化器，而这些都是基于关系型集合理论设计的。拖延到后期

去实现导致了严 重的性能问题。即使对解决了性能问题的那些(或者通过停驻在内存中来掩盖此问题)，也缺少了其他东西，如合适的备份。

我知道一个非常成功的互联网初创公司(你肯定也听过)使用了4个(!!)不同的NoSQL系统来解决问题。

(2014：现在需要什么来解决一切？)

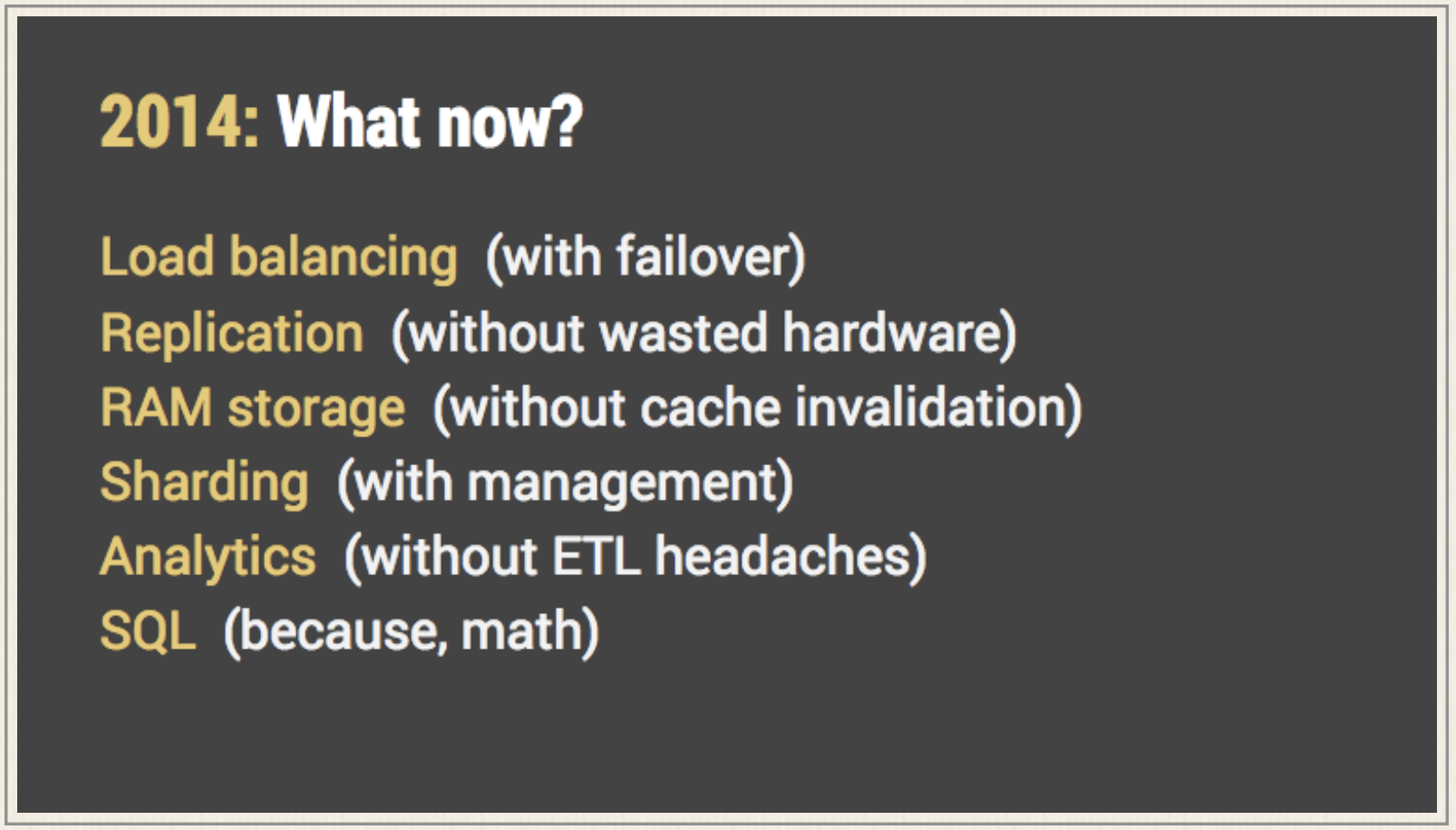
现在已经相当明显，我们不会回到单数据库以及10毫秒一次的随机定位(10-million-nanosecond random seek，上文幻灯有提到，读一次硬盘要10毫秒)的那个从前了。在寻找一劳永逸解决所有问题的炒作周期(hype cycle,也叫技术成熟度曲线)的过程中，有个有趣的模式：聪明的方法在减轻一个痛点的同时会引入新的痛点。



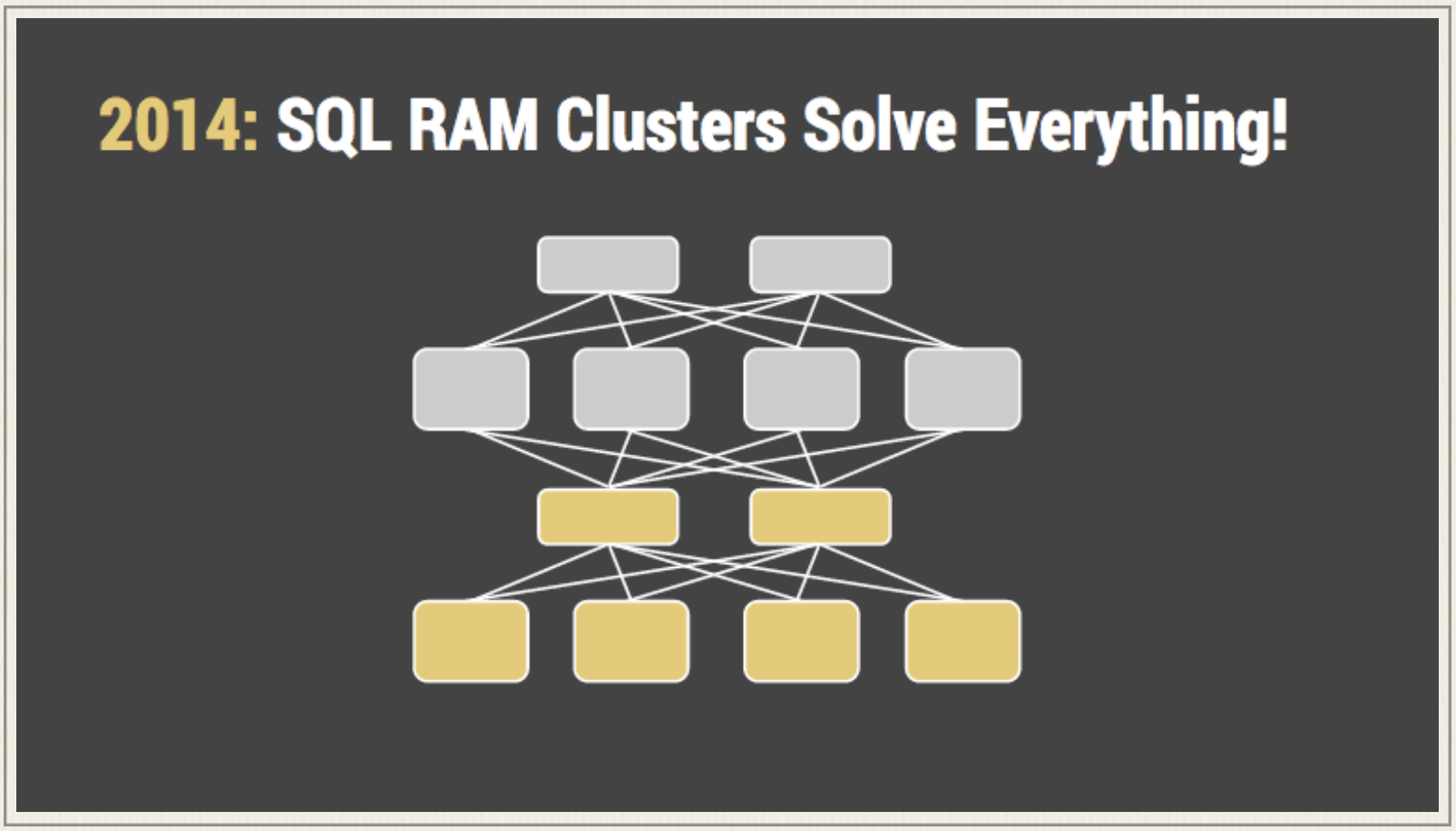
所以下一个添到这张图上的复杂工具是什么呢？也许真正的方法是能简化事情的。

例 如内存：在数据库机器上有很多内存，用做缓冲和计算；Memcached机器上也有很多内存。这些系统中的内存总和至少跟你的工作数据集一样大。如果不 是，你就赚到了(under-bought，低阶买到好货)。而且，我非常怀疑你的缓存层是否100%高效。我打赌你有大量数据在被替换掉之前没有被读取 过，我还打赌你从来没跟踪过。这不意味着你是个坏孩子，而意味着缓存比起其所值，更是个麻烦。

这些组件共有的很多特性看起来，是可以相互组合，并且互补的。只要它们被安排得合理。



一旦你采用下面的公理：系统应该是分布式的，而数据应该是数字化的(solid-state是纯电气的，而不是mechanical机械式的)，有意思的 事情出现了：模型更简单了。在查询触发时才会用到的临时内存数据结构是仅有的结构。随机访问不再是大罪，而是商业的正常过程。你不必担心分页，或者再均衡 (rebalancing)，或者数据的位置。

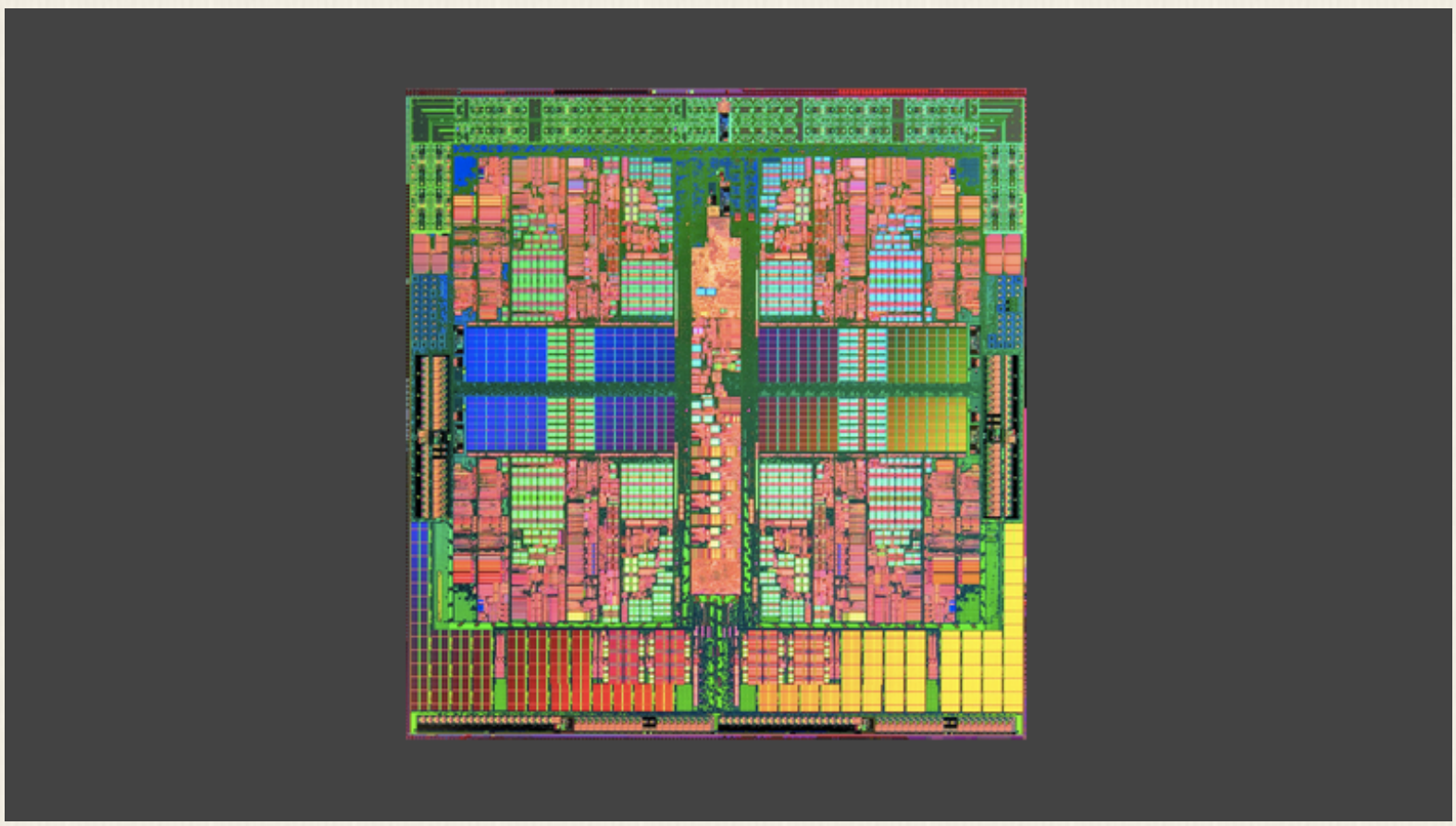


(2014：SQL内存集群解决一切)

这是个优美、简单的架构。就像负载均衡器抽象了应用程序服务器，SQL聚合器(aggregators)抽象了读写的组织细节。把数据存放策略的核心放在稳定的API之下，可以在少量中断的情况下允许两边变化。

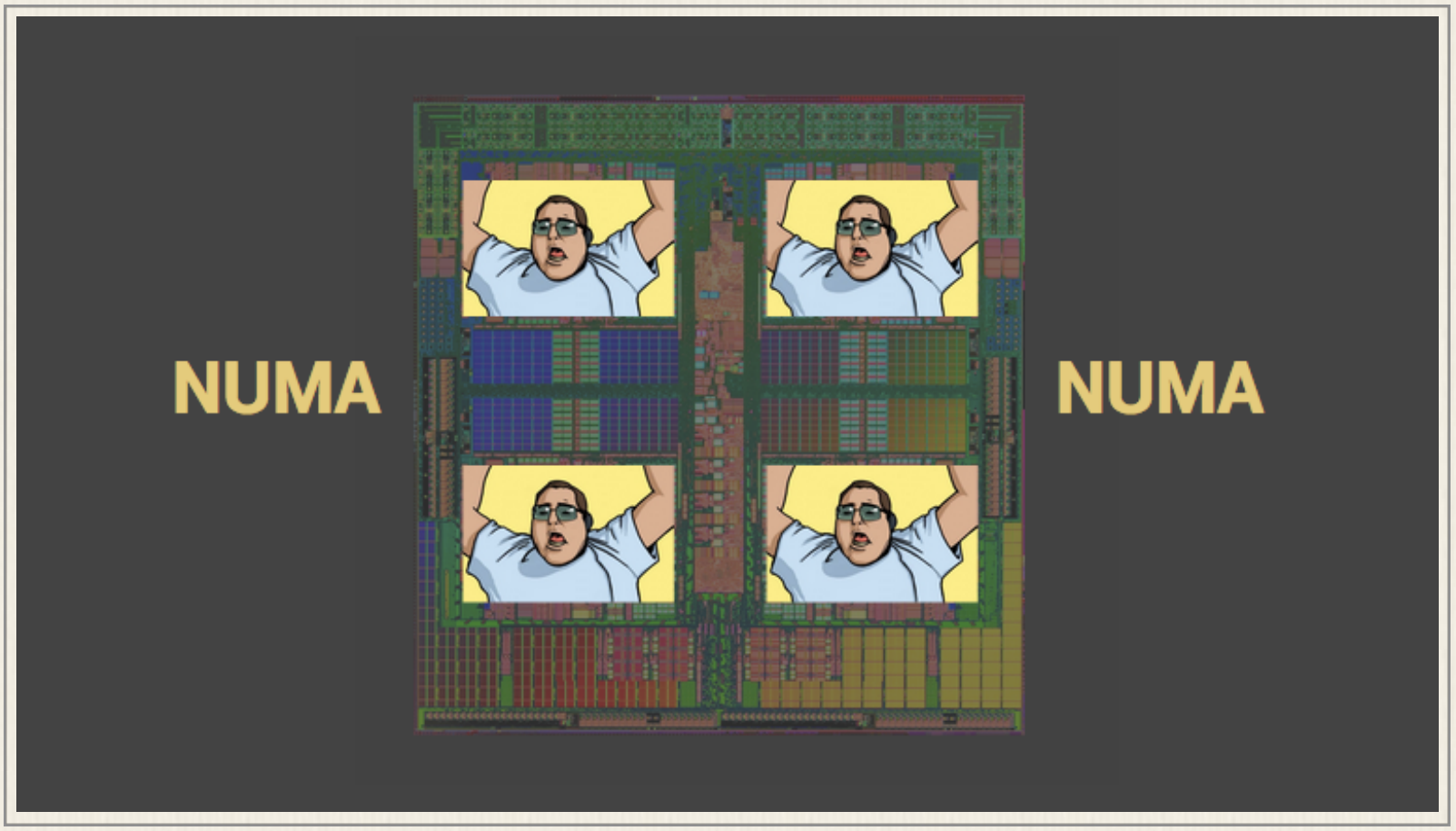
现在，一切都好了，我们最终到达了历史最后的美好之地，对吧?

不管你在何时，对计算艺术状态的自满都是错误的。总会有其他瓶颈。

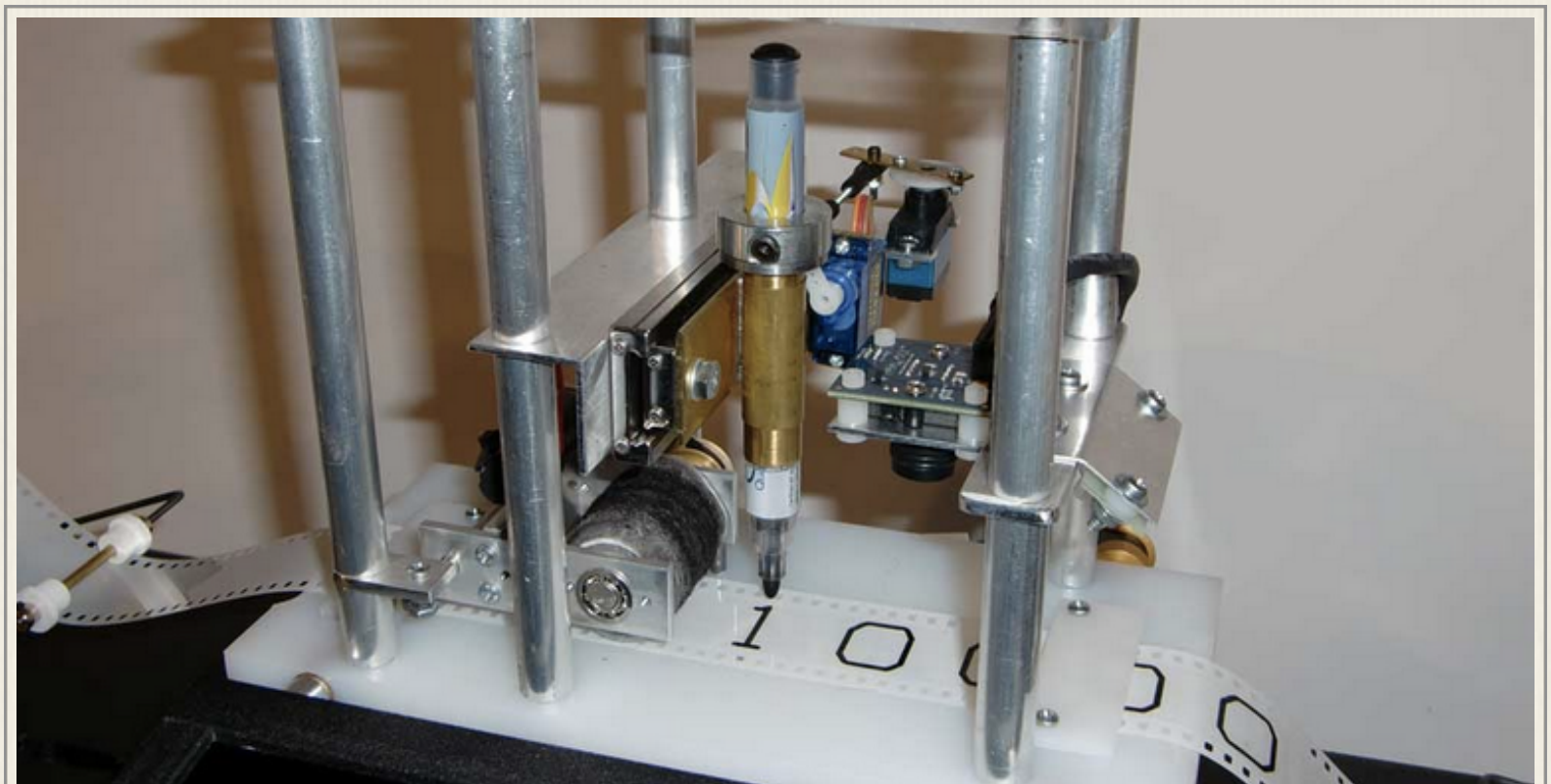


这 是AMD的Barcelona芯片，相当现代化的设计。它有4个核，但是大部分表面都被缓存和核心(core)周围的I/O区域占据，就像WalMart 周围的大型停车场一样。奔腾时代，缓存区域只占晶圆(die)的15%。第三次计算领域的革命在于，CPU相对于内存快了多少。因此晶圆上大片昂贵的区域 都为缓存保留着。

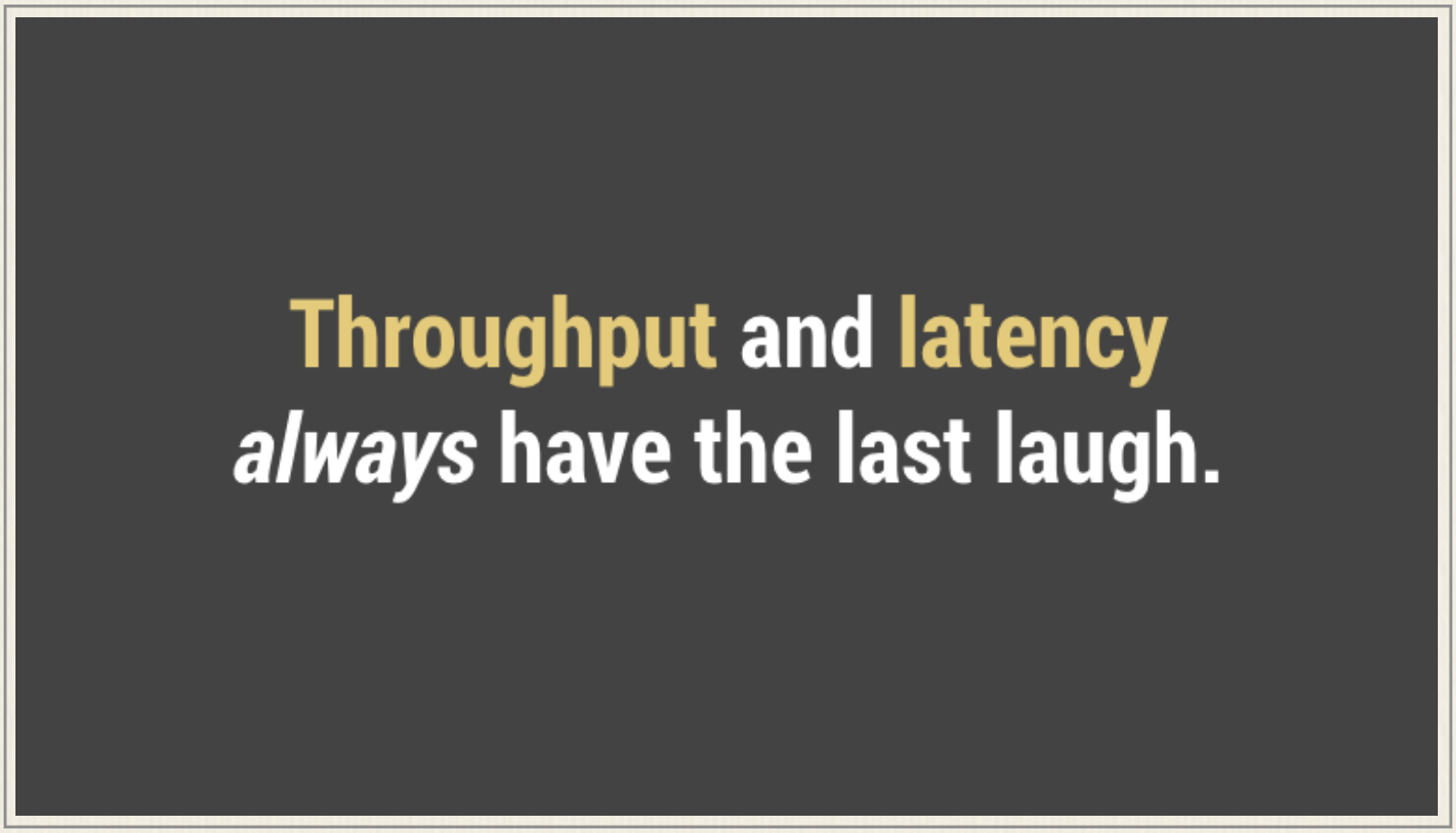
过去，数据库性能的主要关注点在内存和硬盘的延迟，现在我们打趣CPU和内存的延迟不是同样的问题，但是它确实是。



而且我们装作共享内存是存在的，但却不是。有那么多核心和内存，总会有些核心离部分内存很近。



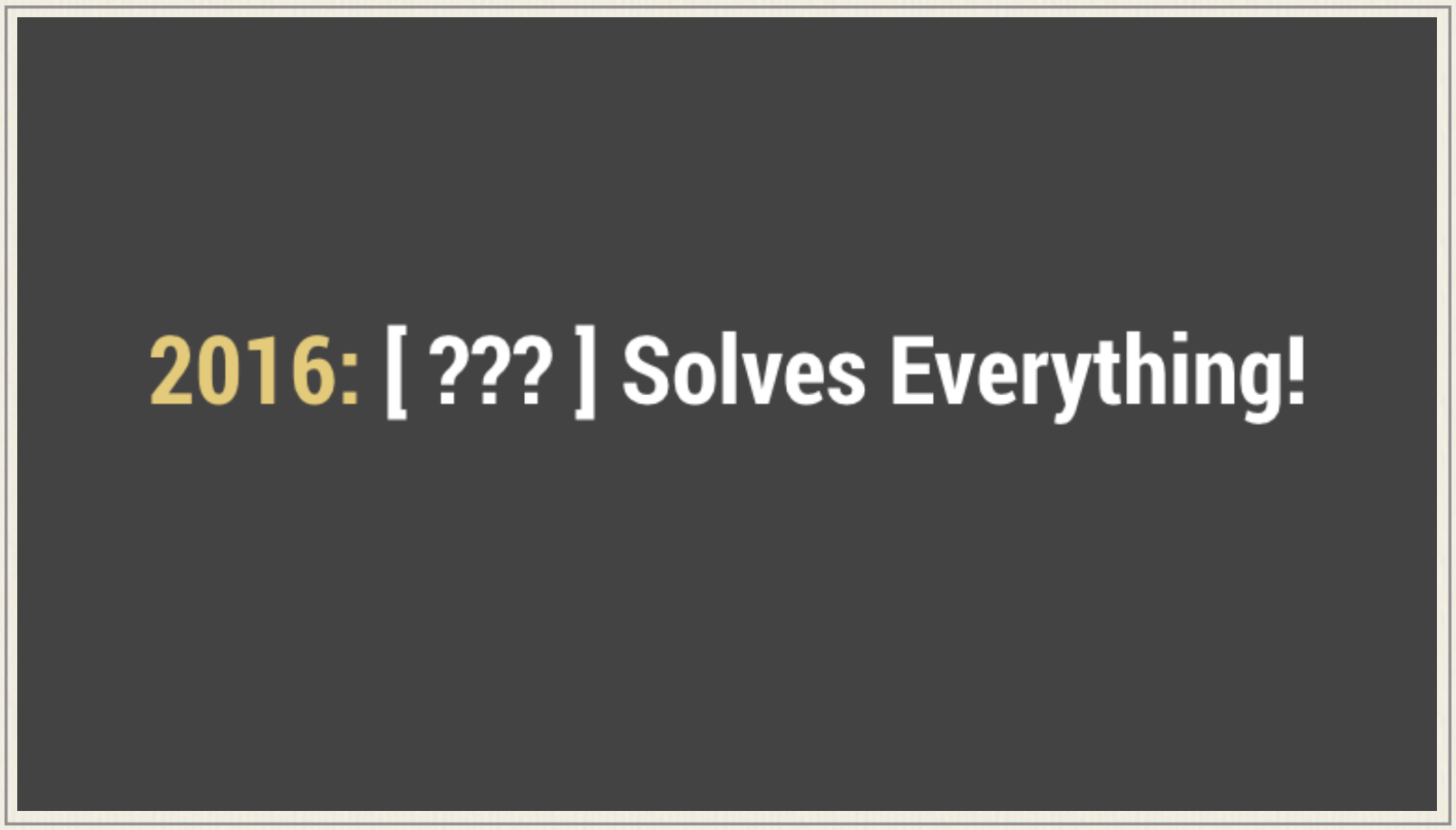
当你仔细想下，计算机确实只做了两件事：读符号，写符号。性能是个计算机有多少数据要移动，数据要到哪里去的功能问题。最好的可能情形是大量的顺序数据流被读取一次，很快地处理后，就不再被用到。GPU是个很好的例子。但是最有意思的负载不是这样的。



(吞吐量和延迟总会笑到最后)

每个随机指针都会保证缓存一次不中，每个对同一块内存区域(比如写锁)的竞争都会引起大量的协调延迟。即使你的CPU缓存命中率达到99%(事实上不可能)，等待内存的时间也会是主导性的。

或者这样说吧：如果磁盘是新的磁带，内存就是新的硬盘，CPU缓存是新的内存。位置仍然有关系。



所 以，什么会解决这个问题？看起来这就是那个相同的古老的矛盾：我们优化随机访问，还是优化串行？我们接纳写，还是读的性能问题？还是我们干坐着等硬件速度 跟上来？也许记忆电阻器(memristor)或者其他技术会使这些问题无关紧要。当然，我也需要些钱(pony, 小马？)。

好消息是分布式数据库的总体物理架构基本成型。数据客户端不再需要处理4或5个不同的子系统的内部细节。这还不完美，也不是主流。但是突破总归需要一段时间来传播。

但如果瓶颈还是在存储位置，这意味着其他部分都成熟了。创新可能在数据结构和算法领域发生。也很少会有清理架构的改动，来承诺一次解决全部问题。如果我们幸运，接下来的15年，SQL数据库会慢慢变得更快更高效，而API是相同的。

但是在此之前，工业界将不会平静。

原译文链接：http://www.oschina.net/translate/cache-is-the-new-ram

比较隐蔽的内存泄露案例分析

作者：百度质量部

1.1. 问题背景

真实项目中的一个待测模块，这里简化一下,整体可以看做是：输入—中间处理–输出模式，如下图1-1

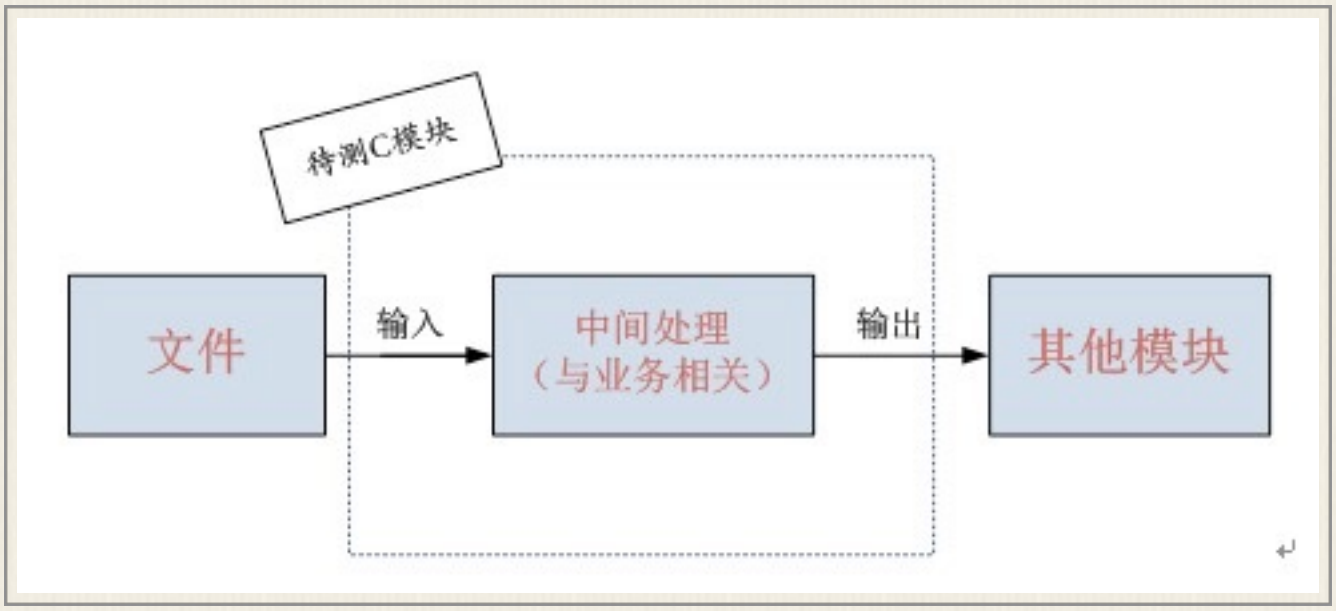


图1-1 待测模块整体框图

其中中间处理与模块业务相关，这里我们可以暂时看做黑盒，不必关心。

1.2. 问题现象

A． 用valgrind跑程序报警图如下1- 2，但是报警所指的地方，研发者坚持认为已经释放，是valgrind误报。

```
==1891==    at 0x490514E: malloc (vg_replace_malloc.c:195).
==1891==    by 0x68FF5C: bsl::syspool::malloc(unsigned long) (bsl_pool.h:42).
==1891==    by 0x6904D6: idl::CarpService_predict_params::create(bsl::mempool*)
(carpservice.idl.h:620).
==1891==                    by    0x6902CE:    SharedTaskManager::createTask()
(SharedTaskManager.cpp:34).
==1891==    by 0x68F52C: PredictJob::run() (PredictJob.cpp:28).
==1891==    by 0x690B20: Thread::startThread(void*) (Thread.cpp:67).
==1891==    by 0x302B806109: start_thread (in /lib64/tls/libpthread-2.3.4.so).
==1891==    by 0x302AFC6002: clone (in /lib64/tls/libc-2.3.4.so).
==1891==.
```

图1-2 valgrind报警图

B． 长期压力测试，未见内存使用持续上升，cpu使用率未见异常；

C．不定时会无故退出：排除人工误操作，日志无异常，无core文件，socket通信已屏蔽SIGPIPE信号。

【NOTICE】在Linux下写socket的程序的时候，如果尝试发送到一个断掉的连接上，就会让底层抛出一个SIGPIPE信号。这个信号的缺省处理方法是退出进程，大多数时候这都不是我们期望的。因此我们需要安全的屏蔽SIGPIPE。

1.3. 追查过程

1）进一步实验的问题现象

这个问题并未引起研发者的注意，研发认为是测试者误操作导致。在程序第一次退出时候,测试者甚至也自我怀疑，是否是误杀了进程或者启动方式有问题。但是测试者在多台机器中启动了程序，观察到了进一步的现象:

A． 都存在文件无故退出问题

B． 退出时间不定，但是都在读完文件之后

C． 观察所有机器的CPU和内存曲线，发现在退出之前都有一个奇怪的波动，波动图案如下图1-3所示：

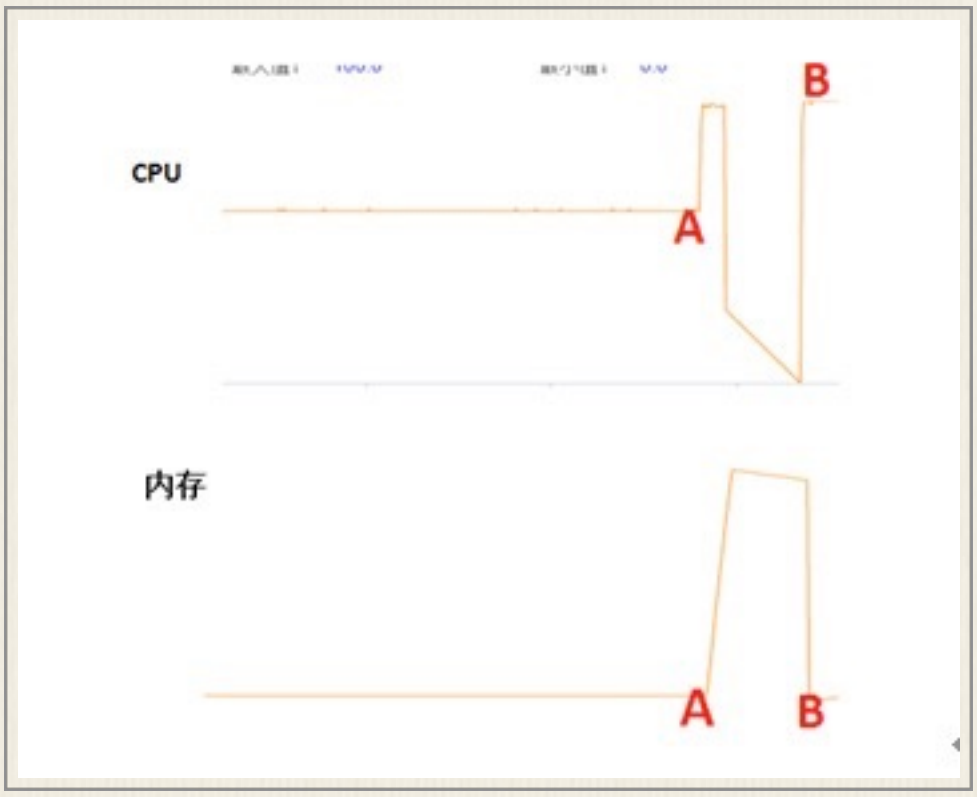


图1-3 程序退出之前CPU和内存曲线

2）由现象想到的…

分析图1- 3的日志文件，找到图中A，B时间点日志文件，分析文件发现：

A时间点：读完文件最后一条记录的时间点；

B时间点：程序退出的时间点。

由此推断：

(1) 文件读完后，未进入中间业务处理逻辑，CPU未参与计算，所以CPU idle一下子升高了

(2) 但是文件读完以后，内存持续上升，用完资源，最后在B点退出，CPU idle和内存使用恢复正常。

3) 从现象看本质

从以上现象和推论，我们可以大胆的猜想：程序存在内存泄露！并且在读文件时候没有泄露，因为长期观察，未见内存持续上涨，内存泄露发生在文件读完以后。

1.4. 验证猜想

根据猜想翻代码：找到如下代码段，如图1-4和1- 5，代码段中我省略了一些与这个bug无关的代码：

图1-4 createTask代码

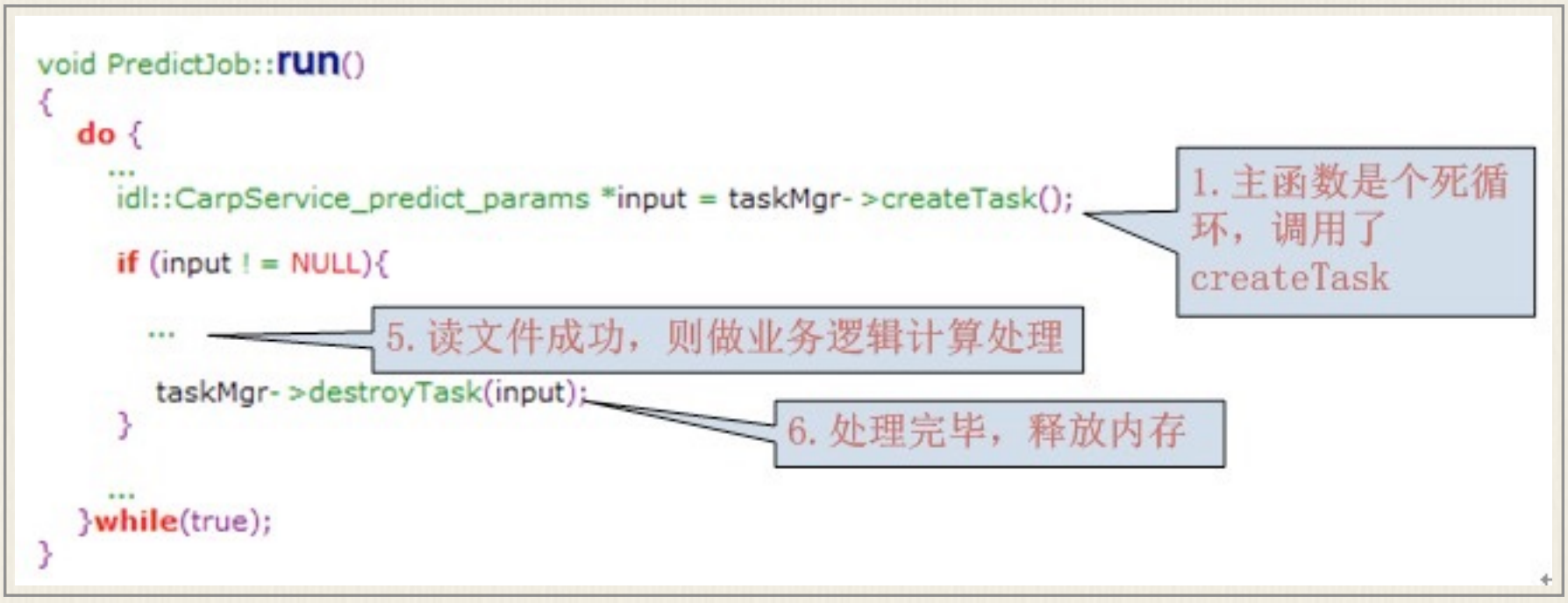


图1-5 主程序死循环调用createTask

【从现象找代码】从代码标注的步骤，一步一步往下读，可以发现：

A. 研发者只释放了读文件成功时候（input!=NULL）申请的input资源（如图1-4标注的第6步）；

B. 文件读失败，返回NULL（如图1-4标注的第三步）；外围函数未释放资源，CreateTask函数里面也未释放input资源；

【从代码看现象】从上图1-3和1-4的代码可以解释，问题现象及图1-3的曲线含义：

A． 当输入大数据文件，性能测试的时候，内存未见持续上升（因为读文件成功，正常释放内存）；

B． 读完文件，readOnce继续读文件执行失败，返回NULL，外面函数检查返回是NULL，不释放input，而creatTask（）死循环不断执行，不断申请内存，不断读文件失败，直到用完内存，程序退出。（所以才会有图1-3的CPU和内存曲线）。

1.5. 如何发现类似问题

A． 认真对待每一次valgrind报警。（事实证明，valgrind所指之处，研发者释放了大部分内存，但在特定场景—即本文中分析的文件读完的场景下，异常处理部分，未释放内存。）

B． 认真观察程序执行期间以及执行完毕后，CPU曲线是否异常，内存是否有持续上涨趋势。

C． 代码评审过程中，需要特别注意正常和异常分支是否有资源泄露的可能。

1.6. 总结:如何避免类似问题

A． 从研发者角度，养成良好的编程习惯，可以将内存分配和释放的过程封装到一个类中，即在构造的时候申请内存，析构的时候释放内存，从而保证没有内存泄露；

B．从测试者角度，代码评审的时候，特别注意：以下函数在资源获取和资源释放时候要对称出现，即在作用域开头申请资源，在作用域末尾释放资源。

malloc/new /new[] 《- -》 free/delete/delete[]

- open/socket/accept/pipe 《- -》 close

fopen/popen 《- -》 fclose

fetch_XXX/get_XXX ßà free_XXX/put_XXX/Release_XXX

*_lock 《- -》 *_unlock

原文链接：http://qa.baidu.com/blog/?p=1209

Kubernetes – Google分布式容器技术初体验

作者：Tim

Kubernetes是Google开源的容器集群管理系统。前几天写的 分布式服务框架的4项特性 中提到一个良好的分布式服务框架需要实现

服务的配置管理。包括服务发现、负载均衡及服务依赖管理。
服务之间的调度及生命周期管理。

由于Kubernetes包含了上述部分特性，加上最近Google新推出的Container Engine也是基于Kubernetes基础上实现，因此最近对Kubernetes进行了一些尝试与体验。

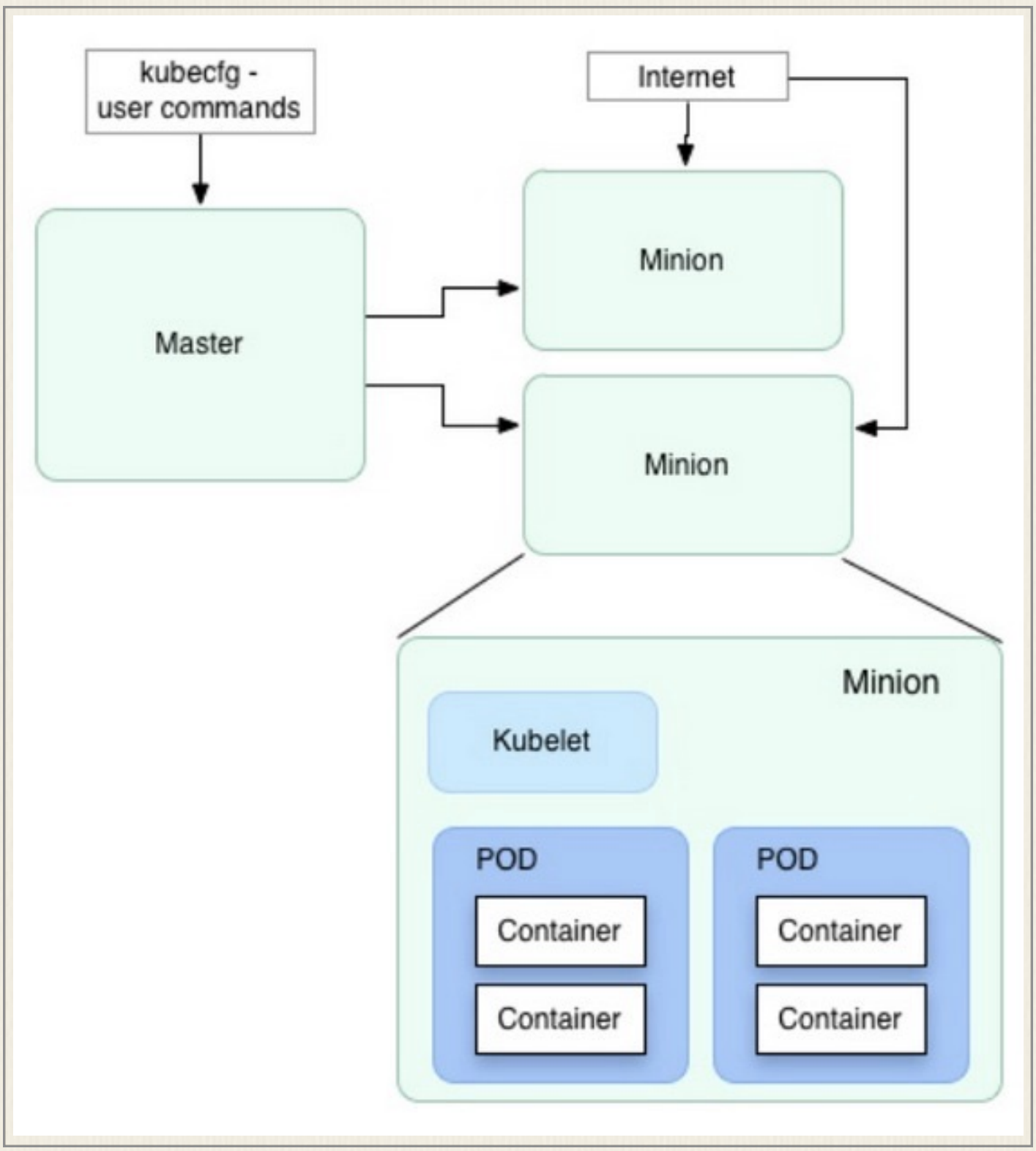
运行环境

Kubernetes目前处于一个快速迭代的阶段，同时它的相关生态圈（比如docker，etcd）也在快速发展，这也意味没有适合新手使用非常顺畅的版本，网上的各种文档（也包括官方文档）和当前最新的发布版会有不同程度滞后或不适用的情况，因此在使用时可能会碰到各种细节的障碍，而且这些新版本碰到的问题，很有可能在网上也搜索不到解决方案。

Kubernetes设计上并未绑定Google Cloud平台，但由于以上原因，为了减少不必要的障碍，初次尝试建议使用GCE作为运行环境（尽管GCE是一个需要收费的环境）。默认的cluster 启动脚本会创建5个GCE instance，测试完需要自己及时主动删除。为了避免浪费，可以将minions减少，同时instance类型选择f1-micro。费用方面一个 f1-micro instance运行1个月大约50元人民币，因此用GCE来测试Kubernetes，如果仅是测试时候开启的话，并不会产生太多费用。

Pods及Replication Controller

Kubernetes的基本单元是pods，用来定义一组相关的container。Kubernetes的优点是可以通过定义一个 replicationController来将同一个模块部署到任意多个容器中，并且由Kubernetes自动管理。比如定义了一个apache pod，通过replicationController设置启动100个replicas，系统就会在pod创建后自动在所有可用的minions中启 动100个apache container。并且轻松的是，当container或者是所在的服务器不可用时，Kubernetes会自动通过启动新的container来保持 100个总数不变，这样管理一个大型系统变得轻松和简单。



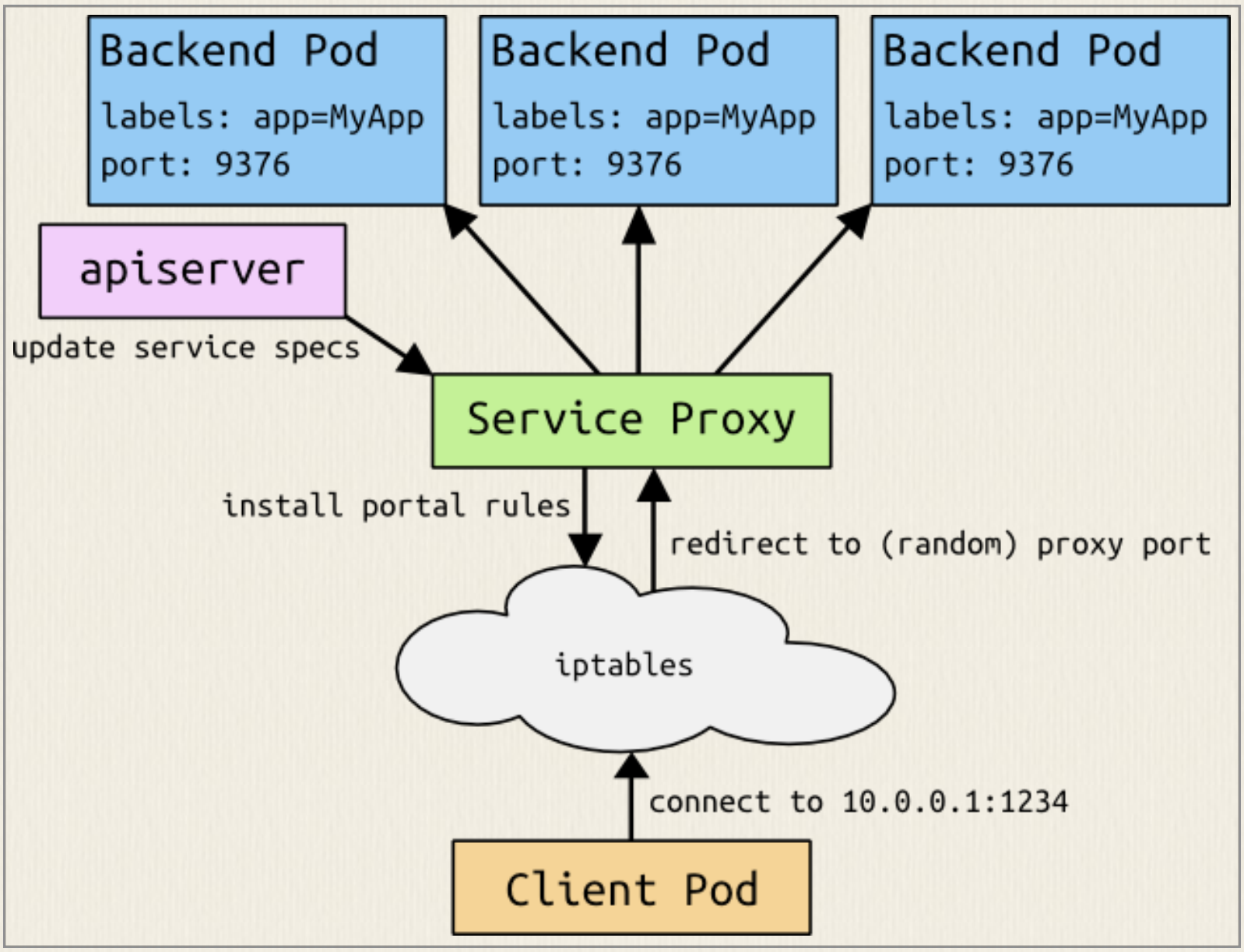
Service 微服务

在解决部署问题之后，分布式服务中存在的一大难题是服务发现（或者叫寻址），用户访问的前端模块需要访问系统内部的后端资源或者其他各种内部的服 务，当一个内部服务通过replicationController动态部署到不同的节点后，而且还存在前文提到的动态切换的功能，前端应用如何来发现并访问这些服务？Kubernetes的另外一个亮点功能就是service，service是一个pod服务池的代理抽象，目前的实现方法是通过一个固定 的虚拟IP及端口来定义，并且通过分布在所有节点上的proxy来实现内部服务对service的访问。

Kubernetes自身的配置是保存在一个etcd（类 似ZooKeeper）的分布式配置服务中。服务发现为什么不通过etcd来实现？Tim的判断更多的是为了Kubernetes上的系统和具体的配置服 务解耦。由于服务发现属于各个系统内部的业务逻辑，因此如果使用etcd将会出现业务代码的逻辑中耦合了etcd，这样可能会让很多架构师望而却步。

尽管没有耦合etcd，部署在Kubernetes中的服务需要通过container中的环境变量来获得service的地址。环境变量虽然简 单，但它也存在很多弊端，如存在不方便动态更改等问题。另外service目前的实现是将虚拟IP通过iptables重定向到最终的pod上，作者也提 到iptables定向的局限性，不适合作为大型服务（比如上千个内部service一起运作时）的实现。

由于service定位是系统内部服务，因此默认情况下虚拟IP无法对外提供服务，但Kubernetes当前版本并没直接提供暴露公网IP及端口的能力，需要借助云服务（比如GCE）的load balancer来实现。



小结

总的看来Kubernetes提供的能力非常令人激动，pod、 replicationController 以及service的设计非常简单实 用。但如果立即将服务迁移到Kubernetes，还需要面对易变的环境。另外尽管Kubernetes提供health check的机制，但service生产环境所需的苛刻的可用性还未得到充分的验证。Service发现尽管不跟Kubernetes的内部实现解耦，但 利用环境变量来实现复杂系统的服务发现也存在一些不足。

安装说明

Kubernetes cluster简单安装说明如下，需要尝试的朋友可参考。

前提准备

一个64 bit linux环境，最好在墙外的，避免访问google cloud出现超时或re-set等问题；另外创建Google Cloud帐号，确保创建instances以及Cloud Storage功能可用；

安装步骤

1. 安装go语言环境（可选，如果需要编译代码则需要）

2. 安装Google cloud sdk

```
$ curl https://sdk.cloud.google.com | bash
$ gcloud auth login
```

按提示完成授权及登录

3. 安装 etcd 二进制版本(V0.4.6), 解压后将其目录加入PATH

4. 安装 kubernetes最新的relase binary版本（V0.5.1）

修改 cluster/gce/config-default.sh，主要是修改以下字段以便节约资源。

```
MASTER_SIZE=f1-micro
MINION_SIZE=f1-micro
NUM_MINIONS=3
```

在kubernetes目录运行

```
$ cluster/kube-up.sh
```

执行成功后会显示 done

5. 测试pod

以上脚本启动了examples/monitoring 下面定义的service，如果尝试启动其它自己的pods，比如启动一个tomcat集群

```
{
  "id": "tomcatController",
  "kind": "ReplicationController",
```

```
"apiVersion": "v1beta1",
"desiredState": {
  "replicas": 2,
  "replicaSelector":{"name": "tomcatCluster"},
  "podTemplate":{
"desiredState": {
  "manifest": {
    "version": "v1beta1",
    "id": "tomcat",
    "containers": [{
      "name": "tomcat",
      "image": "tutum/tomcat",
      "ports": [
      {"containerPort":8080,"hostPort":80}
      ]
      }]
    }
  },
  "labels": {"name": "tomcatCluster"}}
  },
  "labels": {
    "name": "tomcatCluster",
  }
}
```

其中pod的tomcat image可以通过Docker Hub Registry https://registry.hub.docker.com/ 搜索及获取

$ cluster/kubectl.sh create -f tomcat-pod.json

创建成功后通过 cluster/kubectl.sh get pods 来查看它所在minion及ip，可以通过curl或浏览器来访问（请开启GCE防火墙端口设置）。

再定义一个 service

```
{
  "id": "tomcat",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 8080,
  "containerPort": 8080,
  "labels": {
    "name": "tomcatCluster"
  },
  "selector": {
    "name": "tomcatCluster"
  }
}
```

保存为 tomcat-service.json

$ cluster/kubectl.sh create -f tomcat-service.json

检查service启动后的ip及端口，由于service是内部ip，可以在GCE上通过curl来测试及验证。

$ cluster/kubectl.sh get services

6. 关闭cluster

cluster/kube-down.sh

原文链接：http://timyang.net/container/kubernetes-evaluation/

Web渗透练习技巧N则（一）

作者：xia0k

简介

对于我们的生活来说，web的重要性不言而喻，因为这个看起来简单 的几个页面与我们的生活的联系越来越紧密，我们有更多的个人信息由其承载往来于服务器和我们的电脑之间，正因为如此，web的安全也变得越来越重要，越来 越不能被我们忽视。作为一个网络安全的工作者/爱好者，研究web的安全性也变得越来越重要。

那么，接下来的几篇文章，我将陆续给大家介（ban）绍（yun）一些国外的大牛的web渗透的奇技淫巧，当然这其中肯定也会包括一些基本的但是比较容易被我们忽视的点，希望对研究web或者对web有兴趣的同学有点帮助。

前言

练习过程中需要的源文件可以在这里下载

链接: http://pan.baidu.com/s/1i3Dt797 密码: 17fn

环境可能用到的软件有（除了环境软件外，其他软件用到后会有另外的简介）：

XAMPP（Apache）

MySQL

Hydra（win）

John theRipper等

因为内容比较多，一篇文章写完可能有点多，所以本系列分为几篇文章，文章中所需要的。闲话不多说，开搬！

练习一：机密文件探秘

从过去甚至一直到今天，通过隐藏的方式来保护我们的机密文件仍然是一种比较主流的方式。其实这也就意味着除了一般意义上的不让大家知道隐藏地址之外没有加上任何的防护措施，只要我们能找到这个隐藏的地点，也便很容易就可以访问到这些“机密”的数据。

对于一些网站来说，防护的措施一般就是不在主页或者其他子页面上包含任何有关机密页面的地址，以防止被蜘蛛很快捉到从而失去机密性。比如很多网站隐 藏自身的登录页面或者一些不对外开放的svn页面或者git页面等等。其实我们可以通过查看网站的robots.txt文件来查看主站的禁止被爬的内容， 说不定里面就包含了什么不可告人的秘密。

现在，大家就可以上传一些文件到DeepDataHiding，然后通过这种方式尝试找一下了。

（ps：练习一确实有点水...大家可以耐心慢慢看）

练习二：字典在手，天下我有

对于一般的渗透来说，没有了思路之后往往就只剩暴力这一条路了。所以，除了一身好运气之外，有一套高质量的字典就显的尤为重要。因为字典这个东西因人而异，我就不上传我自己的字典了，这里我给大家提供一个字典生成的不错的命令行工具——John the Ripper。

工具的地址：http://www.openwall.com/john/

Git地址：https://github.com/magnumripper/JohnTheRipper

JTR本来是一款密码破解工具，但是这里作者并没有按照其原本的功能来使用它，而是利用这个强大的工具生成字典文件。

这里给出简单的生成介绍，以下是生成命令

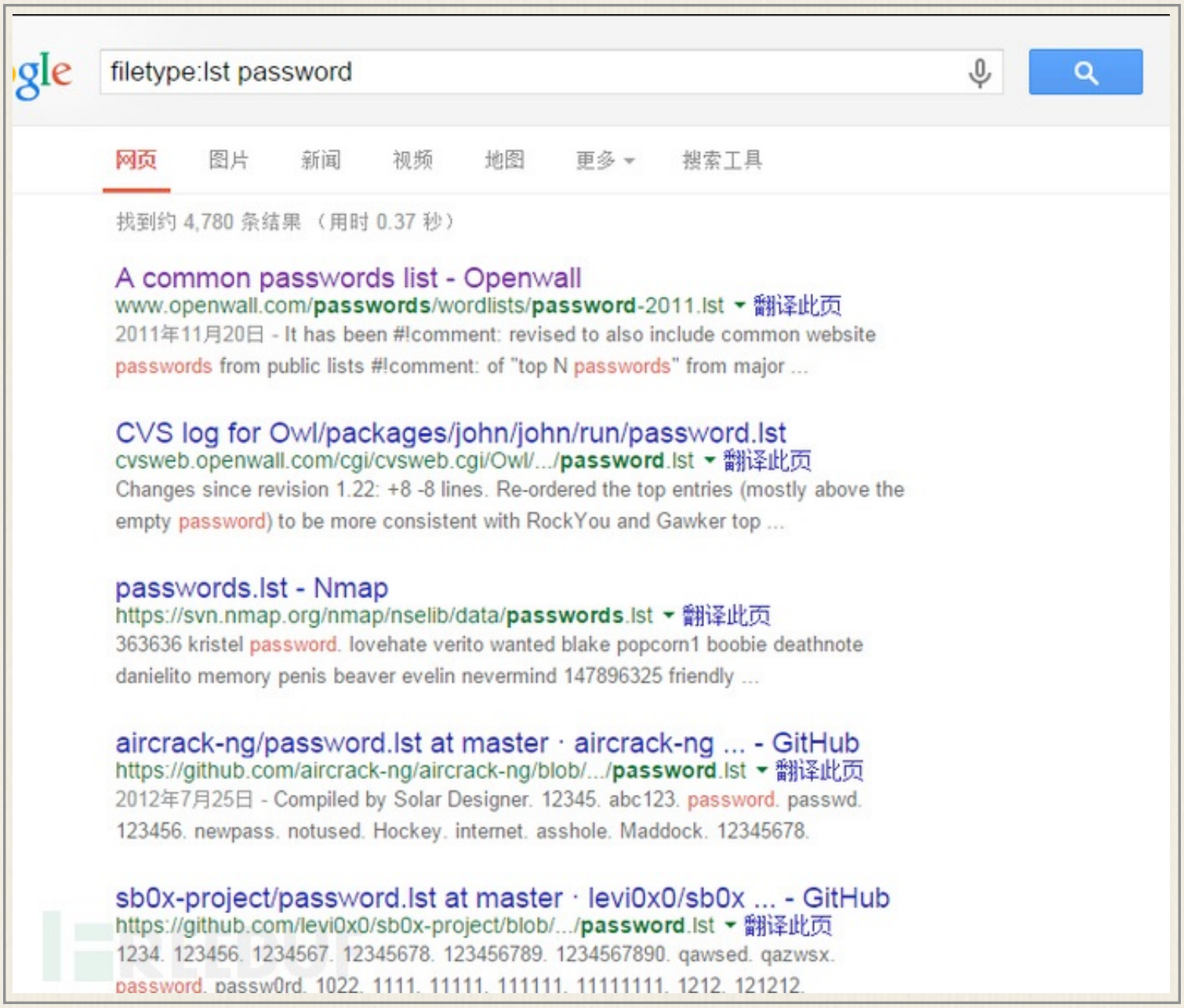
john-mmx–incremental=<mode> –stdout > filename

如果你是kali用户，这里的john-mmx请直接使用john替换，并且无需安装，kali是自带JTR工具的。<mode>可以指定All（包含所有字符），Digit（纯数字），Lanman（字母，数字和一些特殊字符），Alpha（仅字母）。

关于该工具其他的用途（破解），大家可以网上搜一下，或者直接看git的使用说明，如果有需要，我们可以单出一篇文章介绍这款不错的工具。

练习三：善用搜索引擎

这里说的还是Google Hacking，不过这次说的内容是用户名和密码。对于我们需要的用户名和密码，如果自己生成的字典没用，我们倒不如试试用搜索引擎试一下，我们可以用filetype指定搜索类型，比如这里我搜索filetype:lst password，结果如下



当然，我们这里可以根据不同的情况指定不同的关键词，你可能会有意想不到的收获；）

练习四：干掉HTTP认证（HTTPAuth）

在这次的练习中，我们将用到Hydra和我们已经获取到的用户名密码字典文件。

这里还是简单说一下Hydra：

-L <usrlistpath> 指定user字典地址

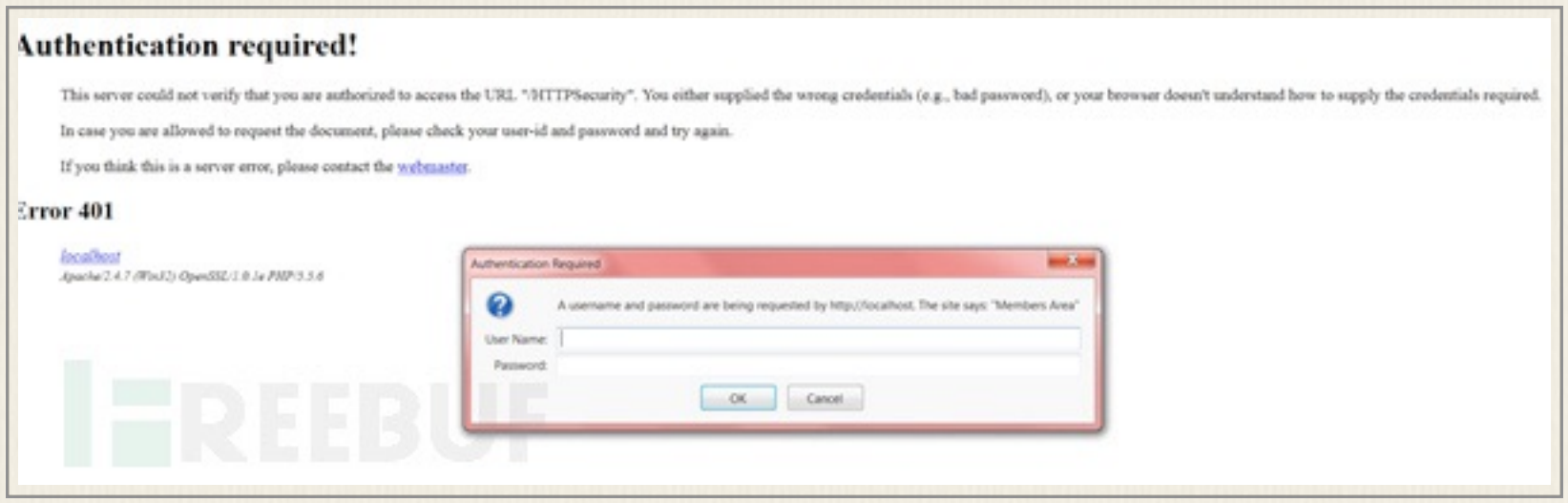
-l <user> 指定单个user

-P <passlstpath> 指定passwd列表地址

-p <password> 指定单个password

接下来，我们需要指定host地址，这里当然是127.0.0.1，然后使用http-get或者http-post指定连接模式，然后是指定用户名和密码地址，最后跟上你访问需要密码的路径，如果没有这里可以指定根目录。整条命令如下所示：

hydra.exe-L D:/WebsiteHacking/FormCracking/usrnames.txt -P D:/WebsiteHacking/FormCracking/passwords.txtlocalhost http-get /HTTPSecurity/



```
[redacted] ~
[redacted] -l admin -P E:/WebsiteHacking/FormCracking/passwords.txt localhost h
ttp-get /HTTPSecurity/
Hydra v8.0 (c) 2014 by van Hauser/THC & David Maciejak - Please do not use in mi
litary or secret service organizations, or for illegal purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2014-08-07 18:07:51
cygwin warning:
  MS-DOS style path detected: E:/WebsiteHacking/FormCracking/passwords.txt
  Preferred POSIX equivalent is: /cygdrive/e/WebsiteHacking/FormCracking/passwor
ds.txt
  CYGWIN environment variable option "nodosfilewarning" turns off this warning.
  Consult the user's guide for more details about POSIX paths:
    http://cygwin.com/cygwin-ug-net/using.html#using-pathnames
[DATA] max 16 tasks per 1 server, overall 16 tasks, 3107 login tries (l:1/p:3107
), ~12 tries per task
[DATA] attacking service http-get on port 80
[80][www] host: 127.0.0.1   login: admin   password: 1234
1 of 1 target successfully completed, 1 valid password found
Hydra (http://www.thc.org/thc-hydra) finished at 2014-08-07 18:07:52

[redacted] ~
```

当然，前提是我们需要先配置一下这个站点让其需要认证。我们需要使用htpasswd.exe在命令行下创建一个密码（具体方式请自行Google）。这里还需要编辑一下.htaccess 文件

AuthTypeBasic

AuthName"Admin Area"

AuthUserFilepathauthorized.htpasswd

Requireuser …

你可以指定一个用户名访问一个页面，同时也可以指定多个不同的用户名访问不同页面的权限，来不断练习这个基础的认证方式。

练习五：干掉POST认证

本次练习所需要的用户名密码字典都在FormCracking文件夹里，因为是测试练习，所以我们默认的密码尽量使用简单一点的。

这里仍然是使用Hydra：

hydra –Lpath/FormCracking/usrnames.txt -P path/FormCracking/passwords.txt 127.0.0.1http-post-form “/FormCracking/index.php:username=^USER^&passwd=^PASS^:Oops”

```
$ hydra.exe -l admin -P E:/WebsiteHacking/FormCracking/passwords.txt 127.0.0.1 h
ttp-post-form "/FormCracking/index.php:username=^USER^&passwd=^PASS^:Oops"
Hydra v8.0 (c) 2014 by van Hauser/THC & David Maciejak - Please do not use in mi
litary or secret service organizations, or for illegal purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2014-08-07 18:12:56
[DATA] max 16 tasks per 1 server, overall 16 tasks, 3107 login tries (l:1/p:3107
), ~12 tries per task
[DATA] attacking service http-post-form on port 80
[80][www-form] host: 127.0.0.1   login: admin   password: qwerty
1 of 1 target successfully completed, 1 valid password found
Hydra (http://www.thc.org/thc-hydra) finished at 2014-08-07 18:12:57
```

与上一个练习不同的是，这条命令指定了一些需要提交用户名密码的字段，这些字段名称主要来自于我们测试网页中需要输入用户名密码的输入栏的属性

另外一点不同是，命令中跟在地址后面的参数与地址之间是需要用冒号(:)隔开的，除此之外，因为我们需要区别密码正确错误的时候的不同的响应，这里 我们指定”Oops”作为当我们登录失败的关键字。当然，这里我们

仍然需要指定^USER^和^PASS^字段以使得我们的字典数据可以填充到请求中。

这里说一下，关于暴力的工具有很多，其中优秀的很多，例如轻量级的Hackbar插件，基于java的Burpsuite等等，但是我们这里力求找一些不同的思路和方法，所以请理性看待。：）

练习六：修改字段

接下来的练习是在ParameterTampering文件夹中进行的。

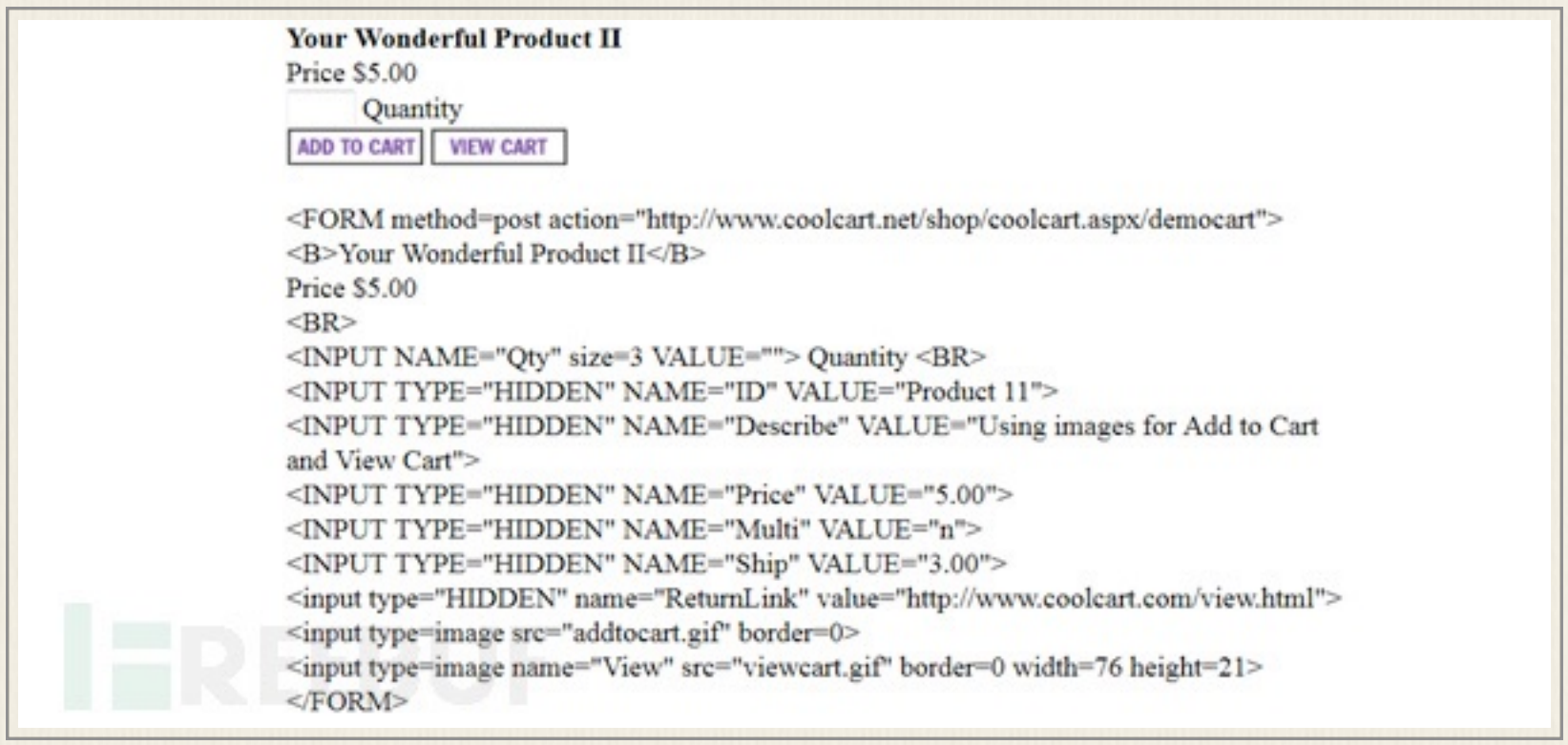
我们会发现在这个文件夹中有三个php文件，这次练习有两个目标，都是要求是在不查看三个php源代码的前提下完成的。

第一个是要求是用浏览器打开login.php然后不需要顾及任何用户名密码正确登录从而正确跳转到member.php页面。

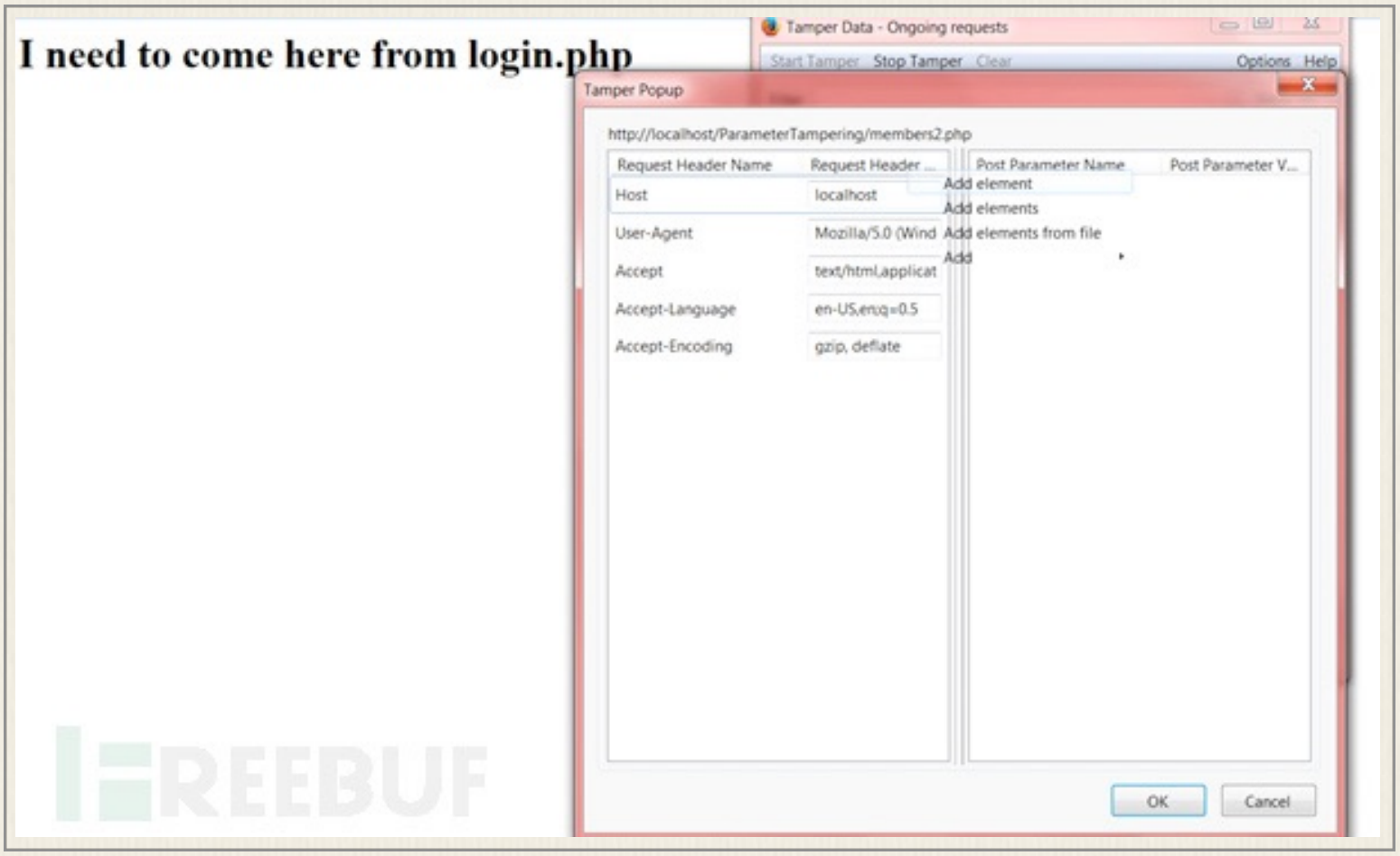
关于这个练习我们给出一点提示：修改页面的字段。

关于这个问题，一开始可能很多人觉得很奇怪，觉得不可能会有页面犯这么蠢的错误，把如此重要的数据直接放在前端只加一个<input type=hide>就算了，但是事实是，确实有人这么干了。举个例子，在很久很久以前，一个古老的第三方网站在输入金额然后要向PalPay跳转 的时候就

将这个重要的数据加了hide然后放在了页面中，然后你想付多少钱就可以改成多少钱，据说没有人改成比原价高的数据付钱。有图为证：



第二个要求是通过修改Referer属性来直接登录member2.php，这里我们可以使用firefox的一个插件，TamperData，当然也可 以使用神器Burpsuite，随大家喜欢。虽然可能只是一个小小的头字段，但是一直到今天，仍有一些小的登录后台是通过这种方式来接受认证的，这一点， 我们还是需要记住，说不定哪天就用到了呢。



练习七：暴力锁定账户

如果我们有一个如下的账户锁定机制（基于PHP/MySQL）：

```
//Connecting to the MySQL database
mysql_connect("localhost","root", "") or die(mysql_error());
 mysql_select_db("userdb") ordie(mysql_error());
//Loading the current number of attempts that the user have used
$attempts= mysql_fetch_array(mysql_query("SELECT attempts FROM users WHERE username= '" . $_POST['username'] . "'"))[0];

//If thelogin credentials are incorrect – add 1 to attempts variable
else if($_POST['pass'] != $info['password']) {
        $attempts +=1;
```

```
        echo "This is your " .$attempts . " attempt!";


    //Stopthe rest of the code from executing if the user have attempted to l
ogin withincorrect details at least three times
    if($attempts > 2) {
        die("</pre>
    <h1>Thisaccount is locked. Contact the administrator atsysadmin@sa
mplesite.com</h1>
    <pre>");
            }


    //Update the attempts column of the particular user in the database

    mysql_query("UPDATEusers SET attempts=" . $attempts . " WHERE us
ername = '" .$_POST['username'] .  "'");
```

如果上面的代码就是我们的登录机制，而我们的登录又是依赖于一个WordPress或者Joomla的一个插件，那么就可能存在恶意的人通过多次输入错误密码来锁定他想要锁定的账户。这样肯定不是我们想要的。

一个解决的措施是锁定登录的IP地址同时仅仅锁定该账户一段时间而不是永久锁定。

这里提供一个解决方式（基于PHP/MySQL）如下：

```
    //folderAccountLockout2

    //Inject SQL code

    CREATETABLE users(

    IDMEDIUMINT NOT NULL AUTO_INCREMENT PRIMARY KEY ,

    usernameVARCHAR( 60 ) ,
```

passwordVARCHAR(60) ,

attemptsTINYINT,

timeTINYINT)

我们如果这个时候再向数据库增加一个账户，那么就可能会是这个样子(因为代码放不开，样式稍加了调整):

$insert= "INSERT INTO users (username, password, attempts, time)

VALUES ('".$_POST['username']."','".$_POST['pass']."', '" . "0" . " , '-1'" . ")";

//attempts

//time whenlockout was set

我们使用数字-1来声明账户未被锁定，代码可以如下所示:

if($attempts > 2) {

// Ifthere no lockout, create one and notify when the account is going to be active

if($info["time"] == "-1") {

$expectedRelease = date("H") + 1;

mysql_query("UPDATEusers SET time=" . date("H") . " WHERE username = '" .$_POST['username'] . "'");

die("</pre>

<h1>Thisaccount is locked. Contact the administrator at sysadmin@samplesite.com"

.". It is going to be active at: ". $expectedRelease . " o'clock</h1>

<pre>

");

}

```
    //Otherwise, remove lockout
        else if ($info["time"] != -1&& date("H") > intval($info["time"])) {
        mysql_query("UPDATE users SETtime='-1' WHERE username = '" .
$_POST['username'] .  "'");
            $attempts = 0;
          }
    else {
    //If theaccount already has locked out and one hour has not passed, jus
t say it islocked and quit
          die("</pre>
    <h1>Thisaccount is locked. Contact the administrator atsysadmin@sa
mplesite.com</h1>
    <pre>
");
        }
```

这是一个简单的账户错误三次之后的锁定机制，当然该锁定也只是锁定一个小时，过了一个小时也就自动解锁了。这段代码也可以在改练习的文件夹中找到。

总结

这篇文章介绍了很多特别基础也特别简单但是容易被我们忽略的点，最后一个练习也是重点在于提出解决措施，因为这篇文章也是作者开题第一篇，所以思路 上也不是特别清晰，不过总的来说对于一个初学者还是大有裨益的。因为来这里看文章的人各个层次的都有，所以权衡再三，这篇文章我还是搬了过来。希望能对有 些人有所帮助。

文章的练习是由浅入深的，大家可以期待接下来的文章了，等不及的同学也可以过去直接看英文原版：http://resources.infosecinstitute.com/category/hacking-2/

原文链接：http://www.freebuf.com/articles/web/52413.html?utm_source=tuicool

直接拿来用！十大Material Design开源项目

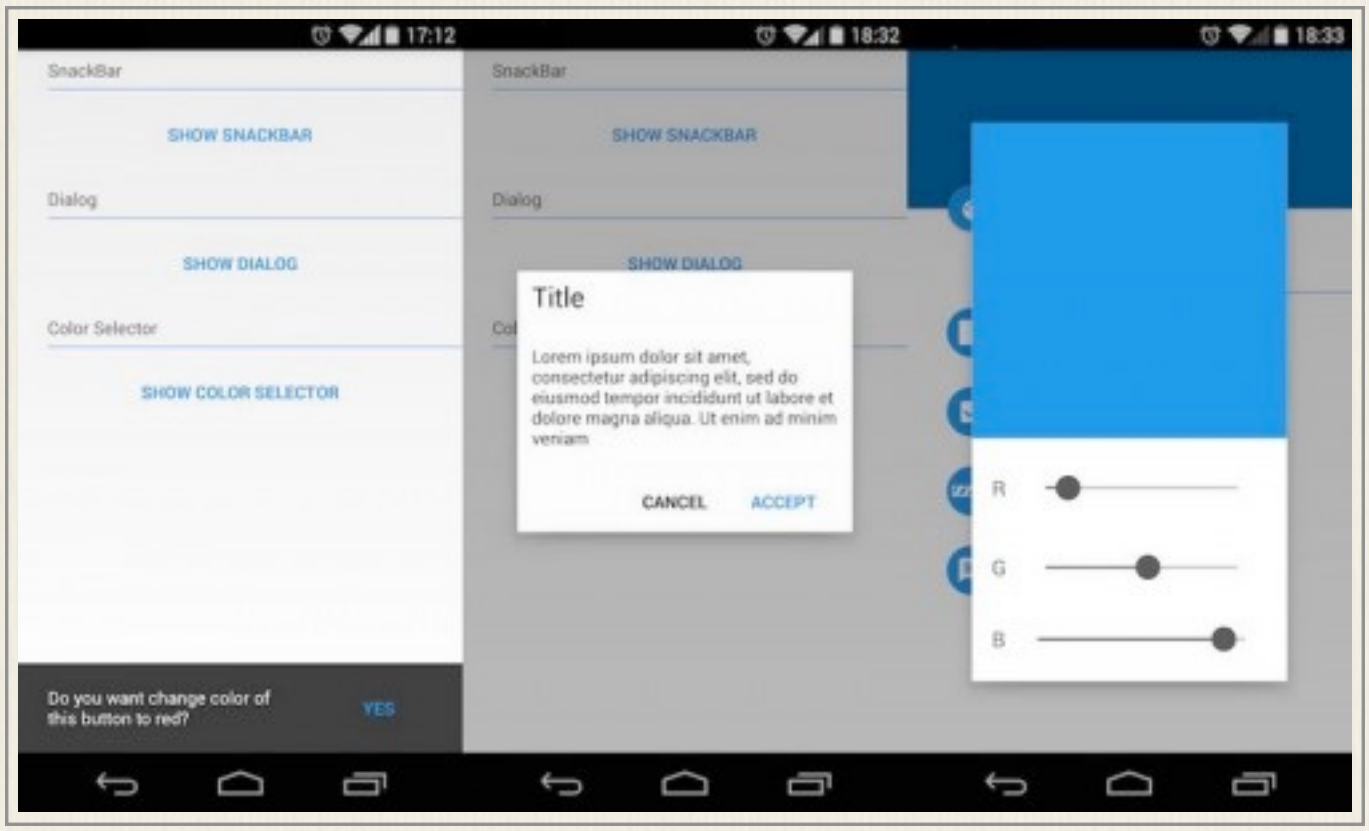
作者：CSDN

介于拟物和扁平之间的Material Design自面世以来，便引起了很多人的关注与思考，就此产生的讨论也不绝于耳。本文详细介绍了在Android开发者圈子里颇受青睐的十个 Material Design开源项目，从示例、FAB、菜单、动画、Ripple到Dialog，看被称为“Google第一次在设计语言和规范上超越了Apple”的 Material Design是如何逐渐成为App的一种全新设计标准。

1. MaterialDesignLibrary

(https://github.com/navasmdc/MaterialDesignLibrary)

在众多新晋库中，MaterialDesignLibrary可以说是颇受开发者瞩目的一个控件效果库，能够让开发者在Android 2.2系统上使用Android 5.0才支持的控件效果，比如扁平、矩形、浮动按钮，复选框以及各式各样的进度指示器等。



除上述之外，MaterialDesignLibrary还拥有SnackBar、Dialog、Color selector组件，可非常便捷地对应用界面进行设置。

进度指示器样式效果设置：

```
<com.gc.materialdesign.views.ProgressBarCircularIndeterminate

        android:id="@+id/progressBarCircularIndeterminate"
            android:layout_width="32dp"
            android:layout_height="32dp"
            android:background="#1E88E5" />
```

Dialog：

```
Dialog dialog = new Dialog(Context context,String title, String message);
dialog.show();
```

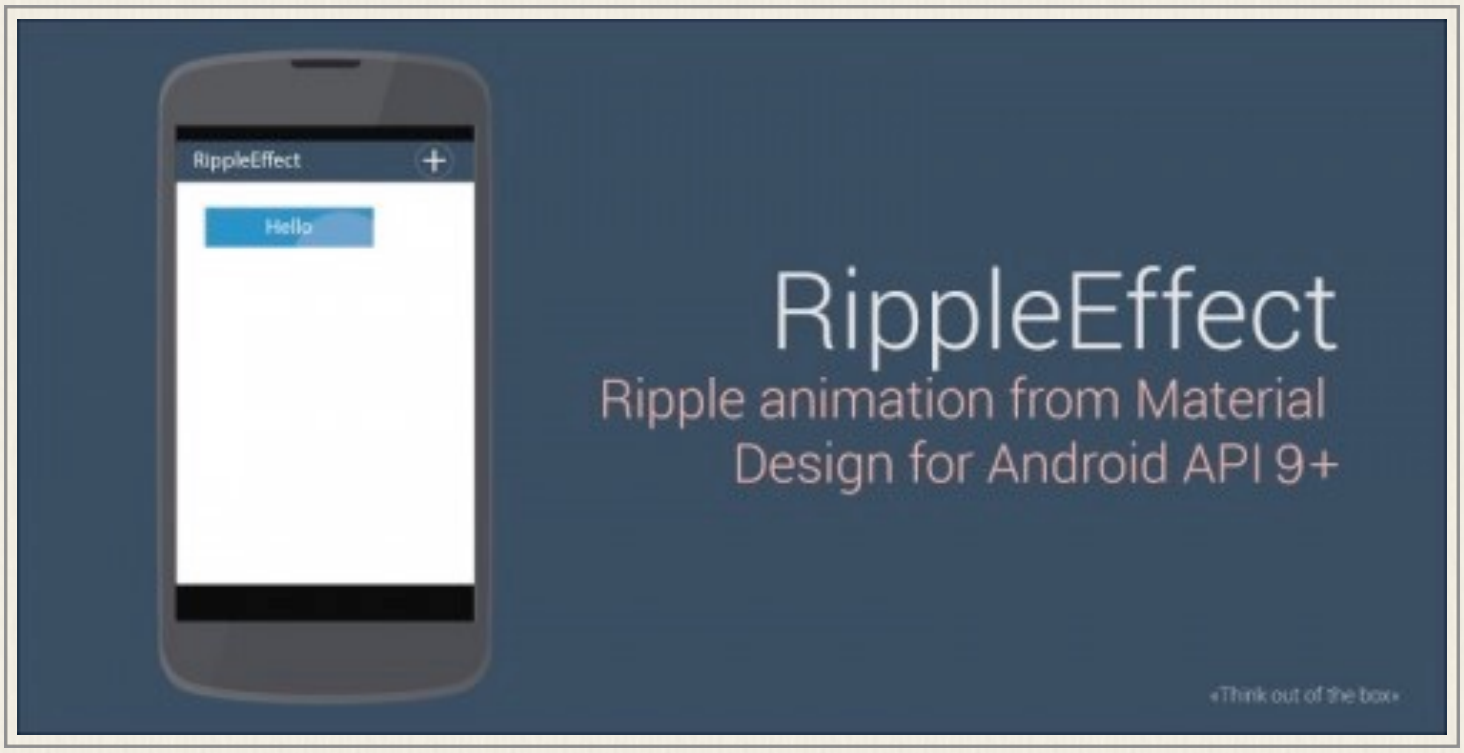
相关链接：MaterialDesignLibrary的mobilehub主页

（http://mobilehub.io/products/materialdesignlibrary）

2. RippleEffect

(https://github.com/traex/RippleEffect)

由来自法兰西的Robin Chutaux开发的RippleEffect 基于MIT许可协议开源，能够在Android API 9+上实现Material Design，为开发者提供了一种极为简易的方式来创建带有可扩展视图的header视图，并且允许最大程度上的自定制。



用法（在XML文件中声明一个RippleView）：

[xml] view plaincopy

```
1.  <com.andexert.library.RippleView
2.    android:id="@+id/more"
3.    android:layout_width="?android:actionBarSize"
4.    android:layout_height="?android:actionBarSize"
5.    android:layout_toLeftOf="@+id/more2"
```

```
6.      android:layout_margin="5dp"
7.      ripple:rv_centered="true">
8.
9.      <ImageView
10.       android:layout_width="?android:actionBarSize"
11.       android:layout_height="?android:actionBarSize"
12.       android:src="@android:drawable/ic_menu_edit"
13.       android:layout_centerInParent="true"
14.       android:padding="10dp"
15.       android:background="@android:color/holo_blue_dark"/>
16.
17.  </com.andexert.library.RippleView>
```

相关链接：RippleEffect的mobilehub主页

(http://mobilehub.io/products/rippleeffect)

3. **MaterialEditText**

(https://github.com/rengwuxian/MaterialEditText)

随 着Material Design的到来，AppCompat v21也为开发者提供了Material Design的控件外观支持，其中就包括EditText，但却并不好用，没有设置颜色的API，也没有任何Google Material Design Spec中提到的特性。于是，来自国内的开发者“扔物线”开发了MaterialEditText库，直接继承EditText，无需修改Java文件即 能实现自定义控件颜色。

自定义Base Color：

1. app:baseColor="#0056d3"

自定义Error Color：

1. app:maxCharacters="10"

2. app:errorColor="#ddaa00"

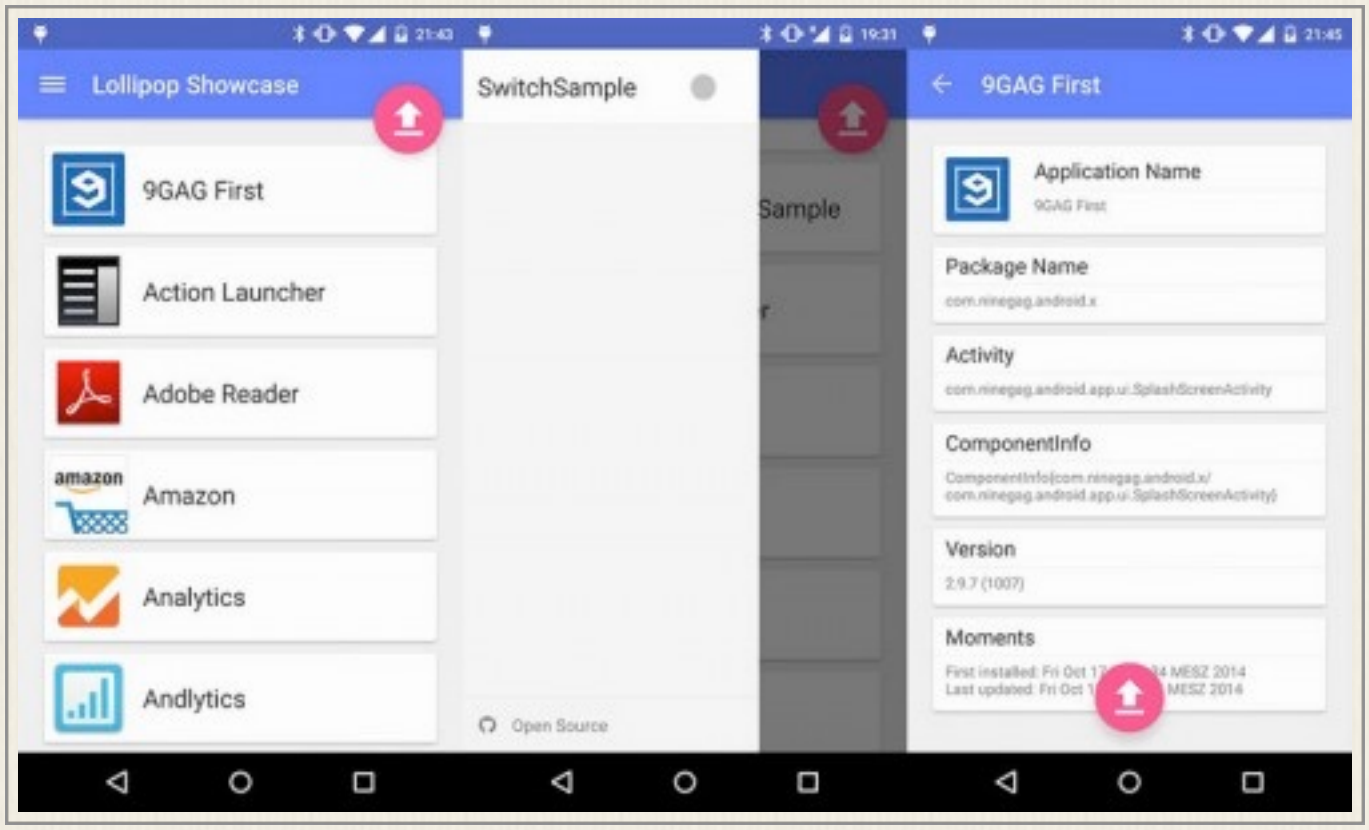
相关链接：MaterialEditText的mobilehub主页

(http://mobilehub.io/products/materialedittext)

4. Android-LollipopShowcase

(https://github.com/mikepenz/Android-LollipopShowcase)

Android- LollipopShow case是由来自奥地利的移动、后端及Web开发者Mike Penz所开发的演示应用，集中演示了新Material Design中所有的UI效果，以及Android Lollipop中其他非常酷炫的特性元素，比如Toolbar、



RecyclerView、ActionBarDrawerToggle、Floating Action Button（FAB）、Android Compat Theme等。

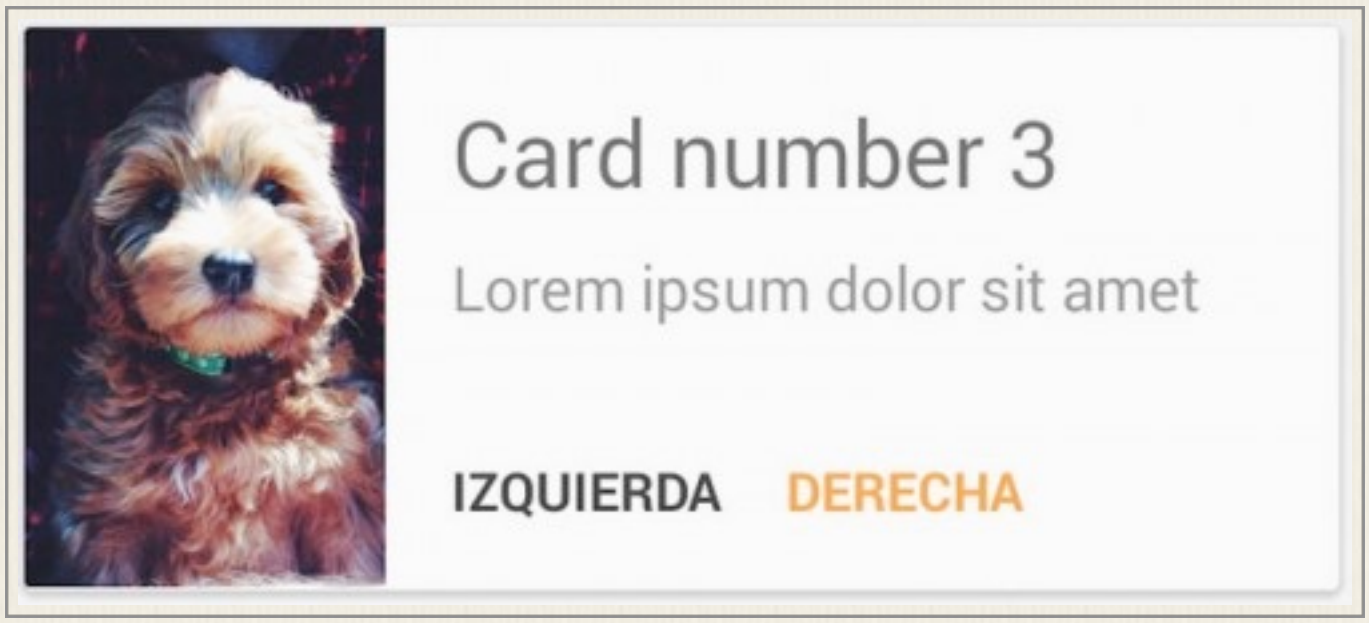
相关链接：Android-LollipopShowcase的mobilehub主页

(http://mobilehub.io/products/android-lollipopshowcase)

5. MaterialList

(https://github.com/dexafree/MaterialList)

MaterialList 是一个能够帮助所有Android开发者获取谷歌UI设计规范中新增的CardView（卡片视图）的开源库，支持Android 2.3+系统。作为ListView的扩展，MaterialList可以接收、存储卡片列表，并根据它们的Android风格和设计模式进行展示。此 外，开发者还可以创建专属于自己的卡片布局，并轻松将其添加到CardList中。



使用过程代码，在布局中声明MaterialListView：

[xml] view plaincopy

```
1. <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2.     android:layout_width="match_parent"
3.     android:layout_height="match_parent"
4.     android:paddingLeft="@dimen/activity_horizontal_margin"
5.     android:paddingRight="@dimen/activity_horizontal_margin"
```

```
6.      android:paddingTop="@dimen/activity_vertical_margin"
7.      android:paddingBottom="@dimen/activity_vertical_margin">
8.
9.      <com.dexafree.materiallistviewexample.view.MaterialListView
10.         android:layout_width="fill_parent"
11.         android:layout_height="fill_parent"
12.         android:id="@+id/material_listview"/>
13.
14. </RelativeLayout>
```

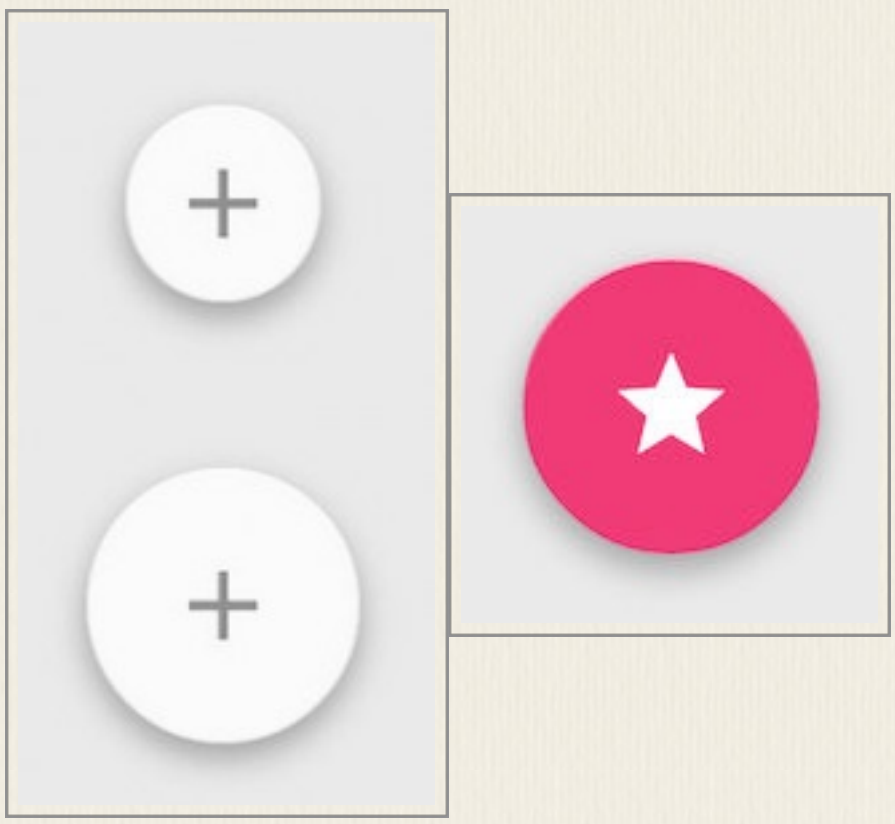
相关链接：MaterialList的mobilehub主页

(http://mobilehub.io/products/materiallist)

6. android-floating-action-button

（ https://github.com/futuresimple/android-floating-action-button ）

Floating Action Button（FAB）是众多专家大牛针对Material Design讨论比较细化的一个点，通过圆形元素与分割线、卡片、各种Bar的直线形成鲜明对比，并使用色彩设定中鲜艳的辅色，带来更具突破性的视觉效 果。也正因如此，在Github上，有着许多与FAB相关的开源项目，基于Material Design规范的开源Android浮动Action Button控件android-floating-action-button便是其中之一。



其主要特性如下：

- 支持常规56dp和最小40dp的按钮；
- 支持自定义正常、Press状态以及可拖拽图标的按钮背景颜色；
- AddFloatingActionButton类能够让开发者非常方便地直接在代码中写入加号图标；
- FloatingActionsMenu类支持展开/折叠显示动作。

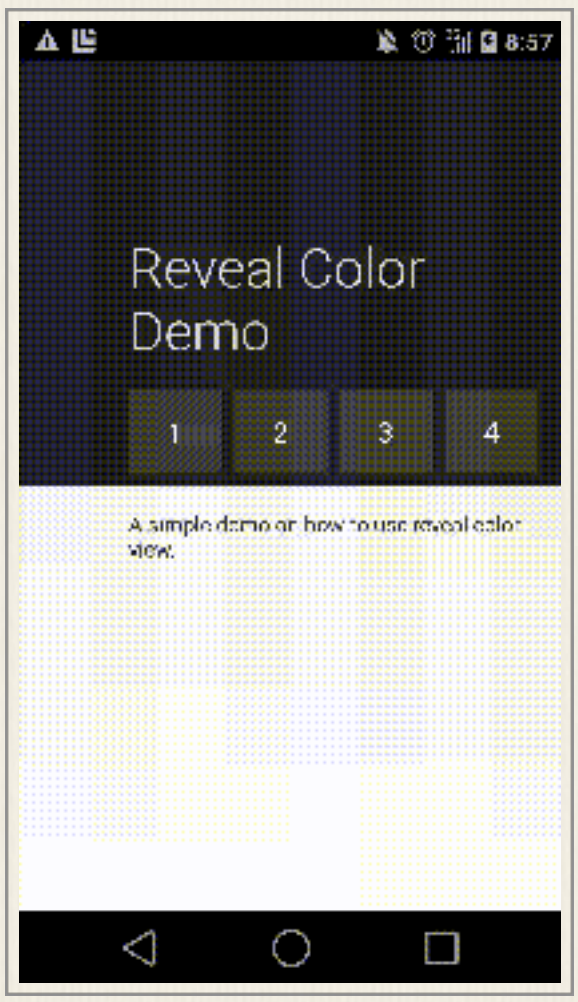
相关链接：android-floating-action-button的mobilehub主页

（ http://mobilehub.io/products/android-floating-action-button ）

7. android-ui

(https://github.com/markushi/android-ui)

android- ui是Android UI组件类库，支持Android API 14+，包含了ActionView、RevealColorView等UI组件。其中，ActionView可使Action动作显示动画效果，而 RevealColorView则带来了Android 5.0中的圆形显示/隐藏动画体验。



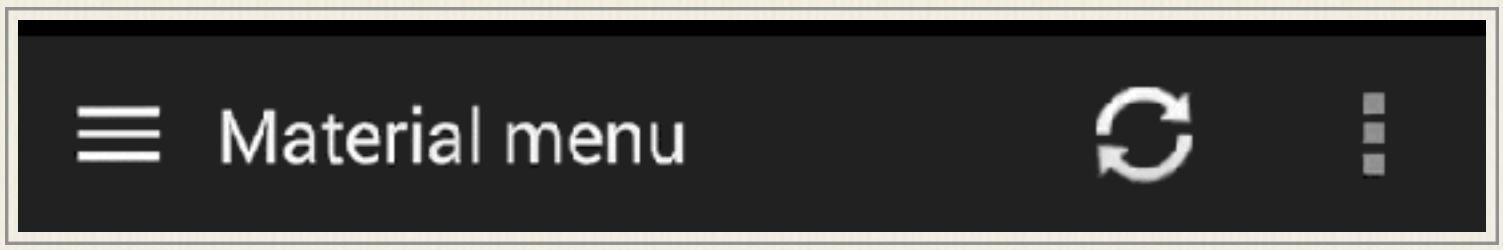
相关链接：android-ui的mobilehub主页

(http://mobilehub.io/products/android-ui)

8. Material Menu

(https://github.com/balysv/material-menu)

Material Menu为开发者带来了非常酷炫的Android菜单、返回、删除以及检查按钮变形，完全控制动画，并为开发者提供了两种MaterialMenuDrawable包装。



自定义颜色等操作:

[java] view plaincopy

```
1.  // change color
2.  MaterialMenu.setColor(int color)
3.
4.  // change transformation animation duration
5.  MaterialMenu.setTransformationDuration(int duration)
6.
7.  // change pressed animation duration
8.  MaterialMenu.setPressedDuration(int duration)
9.
10. // change transformation interpolator
11. MaterialMenu.setInterpolator(Interpolator interpolator)
12.
```

```
13.  // set RTL layout support
14.  MaterialMenu.setRTLEnabled(boolean enabled)
```

相关链接：Material Menu的mobilehub主页

(http://mobilehub.io/products/material-menu)

9. Android-ObservableScrollView

（ https://github.com/ksoichiro/Android-ObservableScrollView ）

Android- ObservableScrollView是一款用于在滚动视图中观测滚动事件的Android库。它能够轻而易举地与Android 5.0 Lollipop引进的工具栏（Toolbar）进行交互，还可以帮助开发者实现拥有Material Design应用视觉体验的界面外观，支持ListView、ScrollView、WebView、RecyclerView、GridView组件。

交互代码回调:

[java] view plaincopy

```
1.  @Override
2.  public void onUpOrCancelMotionEvent(ScrollState scrollState) {
3.      ActionBar ab = getSupportActionBar();
4.      if (scrollState == ScrollState.UP) {
5.          if (ab.isShowing()) {
6.              ab.hide();
7.          }
8.      } else if (scrollState == ScrollState.DOWN) {
9.          if (!ab.isShowing()) {
10.             ab.show();
11.         }
12.     }
```

```
13.     }
```

相关链接：Android-ObservableScrollView的mobilehub主页

(http://mobilehub.io/products/android-observablescrollview)

10. Material Design Icons

（ https://github.com/google/material-design-icons ）

最 后，再来介绍一下Google Material Design规范的官方开源图标集Material Design Icons。良心Google开源了包括Material Design系统图标包在内的750个字形，涵盖动作、音视频、通信、内容、编辑器、文件、硬件、图像、地图、导航、通知、社交等各个方面，适用于 Web、Android和iOS应用开发，绝对是开发者及设计师必备的资源。

图标格式主要包括：

- SVG格式，24px和48px；
- SVG和CSS Sprites；
- 适用于Web平台的1x、2x PNG格式图标；
- 适用于iOS的1x、2x、3x PNG图标；
- 所有图标的Hi-dpi版本（hdpi、mdpi、xhdpi、xxhdpi、xxxhdpi）。

相关链接：Material Design Icons的mobilehub主页

(http://mobilehub.io/products/material-design-icons)

原文链接：http://www.csdn.net/article/2014-11-21/2822753-material-design-libs/1

偏爱MySQL，Nifty使用4个Web Server支撑5400万个用户网站

作者：Todd Hoff

【编者按】Nifty运营网站已经有很长一段时间，而在基于HTML5的WYSIWYG网页制作平台推出后，用户在该公司建立的网站已超过5400 万个，同时其中大部分网站的日PV都不到100。鉴于每个网页的PV都很低，因此传统的缓存策略并不适用。然而即使是这样，该公司也只使用了4个Web Server就完成了这些工作。近日，Wix首席后端工程师Aviran Mordo在“ Wix Architecture at Scale”的演讲中分享了他们的策略，下面我们一起看High Scalability创始人 Todd Hoff的总结：

以下为译文

Wix围绕扩展性上的努力可以用“定制化”三个字来总结——在仔细地审视了系统之后，以高可用和高性能为目标对系统进行了改善。

Wix使用了多数据中心和云服务，这在通常情况下非常少见，他们将数据同时复制到Google Compute Engine和AWS。对于故障转移，他们有专门的应对策略。

从始至终，Wix都没有使用事务。取而代之，所有数据都是不可变的，他们为用例使用了一个非常简单的最终一致性策略。Wix并不是缓存策略爱好者， 简而言之他们并没有打造一个非常高端的缓存层。取而代之，他们将大部分的精力放在了路径渲染优化上，让每个页面的显示时间不超过100毫秒。

Wix开始于一个非常小的系统，使用了单片架构；而在业务发展过程中，他们很自然地过渡到一个面向服务的架构。在整个架构中，他们使用了一个非常成熟的服务识别策略，从而可以很轻易的将所有精力都集中到一个事件上来。

系统统计

- 5400个网站，每个月都会新增100万个
- 800+TB的静态数据，每天1.5TB的新文件
- 3个数据中心+两个云服务（谷歌和亚马逊）
- 300个服务器
- 每天7亿个HTTP请求
- 总计600员工，200人的研发团队
- 系统内服务数量达50个
- 4个公共Web Server来支撑4500万个网站

系统组件

- MySQL
- Google和Amazon云服务
- CDN（内容分发网络）
- Chef

系统衍变

1. 系统始于简单的单片架构，开始时只有一个应用服务器，对于任何人来说，这都是最简单的初始策略，非常灵活且易于更新。

- Tomcat、Hibernate、定制网络框架。
- 使用有状态的登录。

- 无视任何性能和扩展性相关。

2. 两年后。

- 仍然使用单片服务器支撑所有事情。
- 拥有了一定规模的开发团队，且需要支撑一定规模的用户。
- 依赖性产生的问题。某点的改变通常会造成整个系统的变更，无关领域的故障通常会造成整个系统大范围崩溃。

3. 系统拆分的时候到了。

- 到面向服务的架构转变，但是这并不是件容易的事情。比如，你如何将某个功能分离到两个服务中?
- 聚焦用户在系统中的行为，并将之主要归结为3类：修改网站、查看Wix建立的网站以及媒体服务。
- 网站更新包括服务器数据的数据有效性、安全和验证、数据一致性以及大量的数据修改操作。
- 一旦某个网站建立完成，用户就会进行查看。因此，对于整个系统来说，访客的数量十倍于修改者。因此关注点会转换为：

- 高可用性。因为用户业务行为，HA成为系统最大的特性。
- 高性能。
- 高流量值。

- 长尾问题。平台上已经有了大量的网站，但是它们通常都非常小。单独看某个网站，每天可能只有10或100个PV。鉴于这种特性，缓存对于系统扩展来说并不会起到太大的作用。因此，缓存变得非常低效。

- 媒体支撑是第二大服务，包括HTML、javascript、css及images。他们需要一个途径来支撑800TB数据上的大量请求，其中缓存静态内容成为制胜的关键。

- 新系统看起来像一个网络层，网站被切分为3个部分服务：修改部分（任何对数据产生修改的操作），媒体部分（支撑静态内容，只读），公共部分（一个文件被访问的首部分，只读）。

服务打造的指导方针

- 每个服务都有自己独立的数据库，每个数据库只能一个被一个服务写入。

- 数据库只能被服务的API访问，这样可以将关注点分离，并将数据模型对其他服务透明。

- 鉴于性能原因，其他服务只被赋予数据库的只读权限，一个数据库只能被一个服务写入。

- 服务都是无状态的，这让水平扩展非常便捷，业务的增长只需要添加更多服务器就可以支撑。

- 不使用事务。除下billing/financial 事务以外，所有其他服务都不使用事务，这里的理念是避免数据库事务所带来的开销，从而提升性能。鉴于不使用事务，开发者必须考虑设计合适的数据模型来完成事务逻辑特性，从而避免不一致状态。

- 在新服务设计时，缓存并不是所需要考虑的因素。首先，尽可能的考虑服务性能，然后快速的部署到生产环境，查看服务的运行情况。只有在代码无法优化的情况下，才使用缓存来解决性能问题。

更新服务

- 更新服务必须处理大量的文件。
- 数据被使用不可变的JSON pages在MySQL中存储，每天大约250万个。
- MySQL是个非常棒的键值存储。键的设定基于文件的哈希函数，因此键是不可变的，通过主键来访问MySQL可以获得非常理想的性能。
- 可接受的扩展性。在扩展性方面，Wix又做了什么样的权衡？Wix之所以不使用NoSQL的原因是NoSQL往往会牺牲一致性，而通常开发者并不具备处理这种情况的能力，所以坚持MySQL也并非不可。
- 动态数据库。为了给那些经常访问的网站让路，所有网站的冷数据（通常是建立时间超过3个月以上的数据）都会被转移到其他的数据库，这些数据库的性能往往会非常低，但是容量很高。
- 给用户增长留有容量空间。大型档案数据库是非常缓慢的，但是鉴于数据使用的频率这不会出现任何问题。但是一旦这个数据被访问，在下次访问之前这个数据就会被转移到活跃数据库。

打造更新服务的高可用性

- 大数据体积达到一定程度时，任何事情的高可用都是难以保证的。因此，着眼关键路径，在网站领域无疑就是网站的内容。如果网站的一个装饰部分问题，它对网站的可用性不会造成任何致命影响。因此对一个网站来说，关键路径才是唯一关注点。

- 防止数据库崩溃。如果你想尽可能快的完成故障转移，务必做好数据库的备份，并在故障恢复时快速切换到从数据库。
- 数据完整性保护。这里并不一定是恶意破坏，一个bug可能就会对数据存储产生影响。所有数据都是不可变的，为任何数据保存校订版本。最坏的情况下，即使数据被破坏到无法修复，我们也可以将之恢复到修订版本。
- 阻止不可用情况发生。区别于桌面应用程序，网站必须可以被随时随地地访问。因此，在不同地理位置的数据中心，不同云环境中对数据进行备份非常重要，这将赋予系统足够的弹性。

- 在一个网站上点击“保存”按钮，修改会话会给修改服务器发送一个JSON文件。
- 服务器会给活跃MySQL服务器发送页面，同时它会在另一个数据中心进行备份。
- 当数据在本地修改后，一个异步的进程会将修改上传到一个静态网格，也就是所谓的媒体部分。
- 当数据被传输到静态网格后，一个通知会发送给保存在Google Compute Engine上的存档服务。存档服务会连接到这个静态网格，下载这个修改页面，并将之保存在谷歌云服务中。
- 然后，一个通知会发送到修改器，告知页面已经存储到GCE。
- 同时，系统会根据GCE的数据在Amazon中保存另一个副本。
- 当最后一个通知收到后，这意味着这个数据已经保存了3个副本：一个数据库，一个静态网格以及一个GCE。
- 对于新版本来说是3个副本，而对于旧版本来说则会存在两个副本。
- 这个过程具备自我修复的特性。如果这里存在一个错误，当用户下一次更新其网站内容时，所有未完成的修改会被重新上传。

- 停用文件会做垃圾收集处理。

使用无数据库事务方式给数据建模

- 对于服务拥有者来说，他们从来都不期望发生这样的情况：用户同时对两个页面进行修改，结果只有一个页面被存储到了数据库中，这就造成了不一致状态。
- 取得所有JSON文件，随后按照顺序将他们保存到数据库。当所有数据被保存后，一个命令会被发布，它包含了上传到这个静态服务器上所有被保存页面的ID清单（静态服务器中文件名称的哈希值）。

媒体部分

- 存储了大量文件。800TB的用户媒体文件，平均每天300万个文件，5亿条元记录。
- 对图像进行修改。它们会针对不同设备和屏幕对图像进行修改。在这里，可以根据需求插入水印，同时还可以对音频格式进行转换。
- 建立一个一致性分布式文件系统，使用多数据中心备份模式，并且实现跨数据中心的故障恢复。
- 运行的痛苦。32个服务器，每9个月翻一倍。
- 计划迁移到云中以获得更好的扩展性。
- 让供应商锁定见鬼。因为都使用了API，只需要改变实现方式就可以在数周内跨云服务供应商进行迁移。
- 在Google Compute Engine中遭遇失败。当他们从数据中心迁移到GCE时，很快就受到了谷歌云服务的限制。而在谷歌做出了一些改变后，系统得以正常运行。
- 数据是不可变的，因此非常有利于缓存。

- 图像请求会首先发送到CDN。如果所请求的图像在CDN中并不存在，请求会被直接传递给他们奥斯丁的主数据中心。如果在主数据中心也没有发现这个图像，随后寻找的地点就是谷歌云服务。如果谷歌云服务中仍然未发现所请求的图像，那么下一个寻找地点则是坦帕市的数据中心。

公用部分

- 解析URL（在4500万网站中），并将之分配给指定的渲染程序，然后转换成HTML、sitemap XML或者robots TXT等。
- 公用的SLA，峰值时响应时间低于100毫秒。网站必须是高可用的，同时也需要非常高的性能，但是缓存却并不能发挥作用。
- 当一个用户修改某个页面并进行发布后，包括这个页面元素的清单会被推送到公用环境，同时推送的还有路由表。
- 最小化宕机情况。解析一次路由需要促发一个数据库调用。将请求分配个渲染器需要1次RPC调用。获得网站清单也需要一次数据库调用。
- 查询表会在内存中进行缓存，每5分钟修改一次。
- 因为需要传送给编辑器，数据不可能保存为同一种格式。数据使用非规范化格式进行存储，通过主键进行优化，所有需求的内容都会在单一请求中返回。
- 最小化业务逻辑。数据是非规范化的，并且进行预计算。大规模场景下，每秒内发生的每个操作都会乘以4500万次，因此发生在公共服务器上的每个操作都需要被调整。
- 页面渲染

- 由公共服务器返回的html是 bootstrap html类型的，它使用了一个JavaScript Shell，并包含了所有网站清单和动态数据相关的JSON数据。
- 渲染会被放在客户端进行。当下，笔记本电脑和移动设备已经拥有了很强大的性能，它们完全可以从事这个工作。

- 之所以选择JSON，因为解析和压缩都非常方便。
- 客户端上的bug非常容易修补。修补客户端bug只需要重新部署一个客户端代码，如果在服务器端进行渲染，html则会被缓存，因此修补一个bug需要重新渲染上千万个网站。

公用部分的高可用性

- 虽然目标是一直可用，但是总会发生一些意外情况
- 通常情况下：请求由浏览器发出，随之会被传输到一个数据中心，通过一个负载均衡器，它将会给发送到一个公共服务器，解析路由，传 送给渲染器，随后返回到浏览器，并使用浏览器运行javascript。随后，浏览器会对档案服务发送请求，档案服务会做与浏览器相同的操作，然后将数据储存到缓存。
- 数据中心丢失发生的情况：这时候，所有UPS都会挂掉，数据中心也会丢失。所有DNS都会被改变，请求会发送给次数据中心。
- 公用部分丢失的情况：当负载均衡器配置只进行一半发生这个问题时，所有公共服务器都会丢失。或者当部署错误版本时，服务器则会抛 出故障。Wix通过定制负载均衡器代码来解决这个问题，在公共服务器丢失时，他们会将档案服务器路由到高速缓存，即使系统在警报后已经进行故障恢复。
- 在网络连通性很烂的情况：请求由浏览器发出，随之会被传输到一个数据中心，通过一个负载均衡器，并返回对应的html。现在 JavaScript代码必须取回所有的JSON数据和页面。随后进入内容分发网络，发送到静态网格，并获得所有的文件进行网站渲染。在网络很卡的情况 下，文件返回可能无法进行。JavaScript则会做出选择：如果主要位置无法获得文件，代码则会在档案服务中获取。

学到的知识

- 识别业务的关键路劲和关注点，仔细了解产品运行的方式，开发使用场景，尽力让你工作物有所值。
- 使用多云和多数据中心。为了更好的可用性，在关键路径上建立冗余。
- 对数据进行转换，最小化进程外跳，一切只为了性能。预计算并做一切可以做的事情来减少网络抖动。
- 利用好客户端的CPU，为可用性建立关键路径上的冗余。
- 从小做起，先跑起来，然后寻找下一个决策。从始至终，Wix首要解决的都是如何才能让服务可以良好运行的工作，然后有条不紊的转移到面向服务的架构。
- 长尾需要不同的途径进行解决。取代缓存一切，Wix通过优化渲染途径来提升服务，并将数据在活跃和档案数据库中同时进行备份。
- 使用不可变的方式。不可变会对服务的架构产生深远影响，覆盖后端到客户端的所有处理，对于许多问题来说，这都是个优雅的解决方案。
- 供应商锁定根本不存在。所有功能都通过API实现，只需要修改实现就可以在数周内完成不同云供应商的迁移。
- 最大的瓶颈来自数据。在不同云环境中转移大量数据异常困难。

原译文链接：http://www.csdn.net/article/2014-11-21/2822765-nifty-architecture-tricks-from-wix-building-a-publishing-pla

（翻译/童阳 责编/仲浩）

原文链接：http://highscalability.com/blog/2014/11/10/nifty-architecture-tricks-from-wix-building-a-publishing-pla.html

对后端系统规模上升的一些思考

作者：Jerry

随着公司业 务的增长，我们的服务器数量越来越多，上面运行的各种服务也越来越多；系统的架构也在逐渐复杂化，一个业务往往需要调用后端的多个服务才能完成，服务间有 着复杂的依赖关系。这些给我们的运维带来了很大的麻烦，系统发布、监控、扩容等等都随着服务器和服务数量的上升变得越来越麻烦，遇到故障尤其是整个服务器 的硬件故障的时候，恢复时间也越来越长。如果说当前还能忍受的话，那当规模再增长一倍的时候，运维的复杂度就会不可控了。

最近针对这个话题我做了些研究，也有一些思考，还不成熟，但在这里先记录一下。

1. 服务发现

我们当前还是用的最原始的配置文件和 DNS 来做服务发现，Host、端口都是写在配置文件里的，发生变更的时候只能修改配置文件并重启服务。所以当某台机器挂掉的时候，依赖它上面服务的其他系统也 都全部会出问题。而应急的步骤都是先在别的机器上运行新的实例，修改配置文件并重启关联的其他系统。这样做费时、费力、且会有一个时间窗口内系统无法提供服务。

当然我们关键的服务是通过 Nginx 来做了负载均衡/主备的。但这样做还是有两个问题：

1. Nginx 本身成为一个故障点
2. 连接数量翻倍

其中第二个问题曾导致我们的环境出现了 nf_conntrack table full 的问题。我们的关键服务都是多实例负载均衡的，当系统并发上升到一定程度的时候，某些服务器，尤其是跑着 Nginx 的机器很容易出现这个错误。

那如何解决这个问题呢？刚好前些日子在 Tim 大神的博客 上看到了一篇文章为我指明了方向。

文章里提到，目前成熟的分布式服务多使用基于 ZooKeeper 的配置服务来实现的。因此顺着这个思路往下 Google 了一下，发现它是一个靠谱的，可以考虑的方案。

ZooKeeper 本身是个强一致性的分布式配置服务，是分布式系统的基础设施，可以用来实现配置管理、服务发现、Leader 选举、分布式锁等等。Netflix 开源了一个名为 curator(现在是 Apache 的顶级项目）的库，里面实现了不少 ZooKeeper 常用的使用模式，以 Recipes 的方式供用户直接使用。这其中就包含服务发现的功能。

其文档中详细介绍了实现，我这里也简单介绍一下：

1. 每个服务在 ZooKeeper 里都有一个专门的 Path

2. 每个服务实例在启动时都在这个 Path 下注册一个 Node，并附带本实例的 Host 和端口等信息

3. 服务使用者从 ZooKeeper 里查询 Path 下的节点以获取当前活跃的实例和他们的 Host、端口等，使用者可以在客户端作负载均衡

其中服务实例注册的 Node 类型是 ephemeral node，这种类型的节点只有在客户端保持着连接的时候才有效。所以当某个服务实例被停止或者出现网络异常的时候，对应的节点也会被删掉。因此，任何时候 从 ZooKeeper 里查询到的都是当前活跃的实例。借助 ZooKeeper 的推送功能，服务的消费者可以得知实例的变化，从而可以从容应对服务实例的宕机和新实例的添加，无需重启。

总结一下这个方案的好处：

1. 配置的解耦，服务的消费者只需要知道服务在 ZooKeeper 中的注册路径即可，无需配置 Host、端口等（这对于虚拟化或者容器化的方式尤其有用，详见下面 Docker 部分的描述）

2. 客户端的负载均衡，省去了额外的 Nginx，节省连接且去掉了单点

3. 很容易动态增减实例

4. ZooKeeper 本身是一个强一致性的集群，可以做到很高的可用性，消除了单点

比起用 DNS 和配置文件，这个方案优势实在太明显了，我认为是需要迈出去的第一步。

2. 配置管理

我们系统的配置，目前绝大多数用的还是最原始的配置文件方式。对于实例很多的服务，配置管理也是一个很麻烦的事。每次有修改配置的需求时，需要把相关的配置文件全部修改一遍并挨个重启系统。这种做法太过于原始了，成本太高了，且随着实例数的上升线性上升。

引入 ZooKeeper 或者类似的系统，配置管理的问题也可以很自然地得到解决：直接使用它们就好了。易变的配置项全部都注册到 ZooKeeper 中，借助推送，每个服务实例都可以获取到最新的配置。程序只需要保证能动态调整相关的配置参数就行（写成 static final xxx 的都可以改改了）。

我在 Curator 里没有看到现成的配置管理工具，但是借助其 Framework 和 Node Cache 应该很容易自己实现一个。

3. Docker

在实现了上面的目标以后，我认为还可以使用现在红得发紫的新兴技术 Docker 来更进一步简化部署和运维，以便扫清进一步增长的障碍。

Docker 的背后，还是已经存在多年的 Linux 技术如 LXC、Aufs、cgroup 等。借助这些技术，我们可以在一台 Linux 机器（Host）上运行多个互相隔离的容器，它们和 Host 共享内核，但是有自己独立的进程空间、文件系统空间，在容器内的进程看来好像是独立的机器一样。看起来容器和 VMWare，Xen，KVM 等虚拟机差不多，但原理完全不同——前者是虚拟硬件设备，后者只是 Linux 内核做出的隔离而已。容器更加轻量，因此性能

损耗小很多，一台 Host 上可以运行很多的容器。Docker 把这些技术的使用简化了很多，让人可以通过命令很简单地管理容器镜像和运行容器。Docker 借助 Aufs 实现了镜像的『继承』机制，从而节省磁盘空间并且简化镜像的创建和管理。

使用 Docker 能带来下面这些好处：

1. 可以简化开发、测试和部署的环境准备，消除环境变更带来的问题。通过 Dockerfile 可以很容易脚本化镜像的创建。

2. 简化部署。运维不再需要了解应用的细节，直接管理容器即可。

3. 很容易实现自动化。

不过下面这个问题也让我比较顾虑：

Docker 容器的 IP 地址是随机分配的。对于应用本身，通过前面提到的服务发现机制可以消除随机 IP 带来的影响，但是对于一些基础设施如 Nginx，MySQL， Redis 等，可能就是问题。比如 Nginx 中配置的 Upstream 如果重启并且 IP 变化了，Nginx 如何快速获取这个变化并重新配置？ 解决方案倒也有 ，需要借助类似 ZooKeeper 的 etcd。通过类似的思路，我们完全可以统一使用 ZooKeeper 来实现。不过我认为还是比较麻烦，并且 Redis Sentinel、Redis 主从、MySQL 主从能否用类似的方法实现动态发现也是个问题。

4. 结论

对于服务发现和配置管理，我认为是需要迈出的第一步，它可以为更进一步的演进扫清障碍。对于 Dock er，我目前还是稍微持保守态度一些。只要减小应用对环境的依赖，并借助 chef，puppet 之类的配置管理工具，大规模的部署应该一样不会太复杂。

5. 参考

- Curator Service Discovery 实例：
http://aredko.blogspot.jp/2013/10/coordination-and-service-discovery-with.html
- 开源服务发现方案（有ZooKeeper之外的方案）：
http://jasonwilder.com/blog/2014/02/04/service-discovery-in-the-cloud/
- Pinterest 对 ZooKeeper 服务发现的改进（主要防止 ZooKeeper 本身出故障）：
http://engineering.pinterest.com/post/77933733851/zookeeper-resilience-at-pinterest
- Docker 核心技术概览：http://www.infoq.com/cn/articles/docker-core-technology-preview

原文链接：http://jerrypeng.me/2014/11/24/thoughts-about-backend-growth/